1. Preface
2. LabVIEW Programming Environment
    1. LabVIEW Programming Environment
    2. Lab 1: Introduction to LabVIEW
3. LabVIEW MathScript and Hybrid Programming
    1. LabVIEW MathScript and Hybrid Programming
    2. Lab 2: LabVIEW MathScript and Hybrid Programming
4. Convolution and Linear Time-Invariant Systems
    1. Convolution and Linear Time-Invariant Systems
    2. Lab 3: Convolution and Its Applications
5. Fourier Series
    1. Fourier Series
    2. Lab 4: Fourier Series and Its Applications
6. Continuous-Time Fourier Transform
    1. Continuous-Time Fourier Transform
    2. Lab 5: CTFT and Its Applications
7. Digital Signals and Their Transforms
    1. Digital Signals and Their Transforms
    2. Lab 6: Analog-to-Digital Conversion, DTFT and DFT
8. Analysis of Analog and Digital Systems
    1. Analysis of Analog and Digital Systems
    2. Lab 7: System Response, Analog and Digital Filters
9. References

Preface

A typical undergraduate electrical engineering curriculum includes a signals and systems course during which students are initially exposed to signal processing concepts such as convolution, Fourier series, Fourier transform and filtering. Laboratory components of signals and systems courses are primarily based on textual .m files. Although the ability to write textual codes is an important aspect of a lab component, students can enhance their understanding of signal processing concepts in these courses if they interactively experiment with their codes.

Our motivation for writing this book has thus been to present an interactive programming approach as an alternative to the commonly practiced textual programming in signals and systems labs to provide an efficient way for students to interact and experiment with their codes. The interactivity achieved via hybrid programming, that is, a combination of textual and graphical programming, offers students a more effective tool to better understand signal processing concepts.

Textual programming and graphical programming both have pros and cons. In general, math operations are easier to code in textual mode. On the other hand, graphical programming offers an easy-to-build interactive and visualization environment along with a more intuitive approach toward building signal processing systems.

To bring together the preferred features of textual and graphical programming, we have designed the labs associated with a typical signals and systems course by incorporating .m files into the National Instruments LabVIEW graphical programming environment. This way, although students program the code in textual .m files, they can easily achieve interactivity and visualization in LabVIEW by just having some basic knowledge of the software. The first two labs provide an introduction to LabVIEW and MathScript (.m files) to help students become familiar with both graphical and textual programming in case they have not already done so in their earlier courses.

In addition to the signal processing concepts, students cover example applications in each lab to learn how to relate concepts to actual real-world

applications. The applications considered span different signal processing areas including speech processing, telecommunications and digital music synthesis. These applications provide further incentive for students to stay engaged in the labs.

The chapters in this book are organized into the following labs:

1. Introduction to LabVIEW

Students gain some basic familiarity with LabVIEW, such as how to use controls, indicators and other LabVIEW graphical features, to make .m files more interactive.

2. Introduction to MathScript

If not already familiar with .m file coding, students learn the basics of this coding.

3. Convolution and Linear Time-Invariant Systems

Students experiment with convolution and linear time-invariant (LTI) systems. Due to the discrete-time nature of programming, students must make an approximation of the convolution integral. The lab, which covers convolution properties, shows how to perform numerical approximation of convolution. To apply convolution concepts, students examine an RLC circuit, and build and analyze an echo cancellation system.

4. Fourier Series and Its Applications

Students explore the representation of periodic analog signals using Fourier series and discuss the decomposition and reconstruction of periodic signals using a finite number of Fourier coefficients. To apply the concepts they have learned, students perform an RLC circuit analysis using periodic input signals.

5. Continuous-Time Fourier Transform and Its Applications

Students implement continuous-time Fourier transform (CTFT) and its properties, as well as cover amplitude modulation and high-frequency noise

removal as CTFT applications.

6. Digital Signals and Their Transforms

Students explore the transforms of digital signals. In the first part of the lab, students examine analog-to-digital conversion and related issues including sampling and aliasing. In the second part, students cover the transformations consisting of discrete Fourier transform (DFT) and discrete-time Fourier transform (DTFT) and compare them to the corresponding transforms for continuous-time signals, namely Fourier series and CTFT, respectively. Students also examine applications such as dual-tone multi-frequency (DTMF) signaling for touch-tone telephones and dithering to decrease signal distortion due to digitization.

7. Analysis of Analog and Digital Systems

During the final lab, students implement the techniques and mathematical transforms they learned in the previous labs to perform analog and digital filtering. They build and analyze a square root system and a filtering system with interactive capabilities.

The codes and files associated with the labs in this book can be downloaded from the website at **www.utdallas.edu/~kehtar/signals-systems(username = signals-systems, password = laboratory).** Note that this book is meant only as an accompanying lab book to signals and systems textbooks and should not be used as a substitute for these textbooks.

We would like to express our gratitude to National Instruments, in particular its Academic Marketing Division and Mr. Erik Luther, for their support and initial publication of this book through lulu.com. We hope its publication now through Connexions would facilitate its widespread use in signals and systems laboratory courses.

Nasser Kehtarnavaz

Philipos C. Loizou

Mohammad T. Rahman

LabVIEW Programming Environment
This chapter provides an introduction to LabVIEW graphical programming.

The LabVIEW graphical programming environment can be used to design and analyze a signal processing system in a more time-efficient manner than with text-based programming environments. This chapter provides an introduction to LabVIEW graphical programming. Also see [link], [link], and [link] to learn more about LabVIEW graphical programming.

LabVIEW graphical programs are called virtual instruments (VIs). VIs run based on the concept of dataflow programming. This means that execution of a block or a graphical component is dependent on the flow of data, or, more specifically, a block executes after data is made available at all of its inputs. Block output data are then sent to all other connected blocks. With dataflow programming, one can perform multiple operations in parallel because the execution of blocks is done by the flow of data and not by sequential lines of code.

Virtual Instruments (VIs)

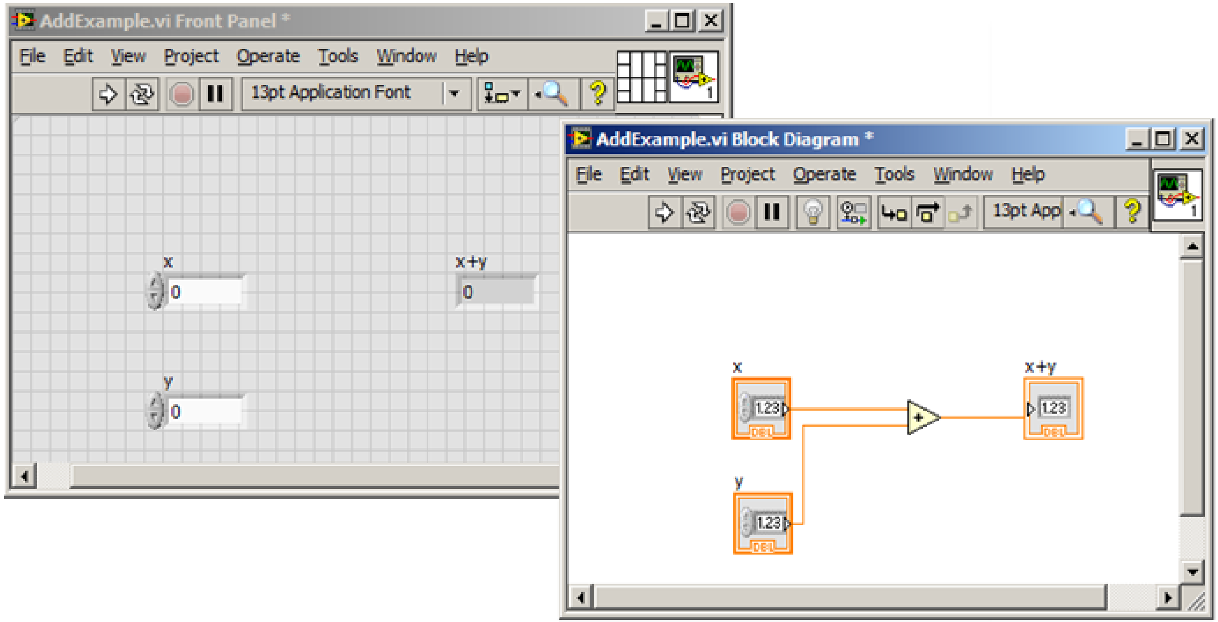
A VI consists of two major components: a front panel and block diagram. A front panel provides the user interface of a program while a block diagram incorporates its graphical code. When a VI is located within the block diagram of another VI, it is called a subVI. LabVIEW VIs are modular, meaning that one can run any VI or subVI by itself.

Front Panel and Block Diagram

A front panel contains the user interfaces of a VI shown in a block diagram. VI inputs are represented by controls such as knobs, pushbuttons and dials. VI outputs are represented by indicators such as graphs, LEDs (light indicators) and meters. As a VI runs, its front panel provides a display or user interface of controls (inputs) and indicators (outputs).

A block diagram contains terminal icons, nodes, wires and structures. Terminal icons, or interfaces through which data are exchanged between a

front panel and a block diagram, correspond to controls or indicators that appear on a front panel. Whenever a control or indicator is placed on a front panel, a terminal icon gets added to the corresponding block diagram. A node represents an object or block that has input and/or output connectors and performs a certain function. SubVIs and functions are examples of nodes. Wires establish the flow of data in a block diagram, and structures control the flow of data such as repetitions or conditional executions. [link] shows front panel and block diagram windows.



LabVIEW Windows: Front Panel and Block Diagram

Icon and Connector Pane

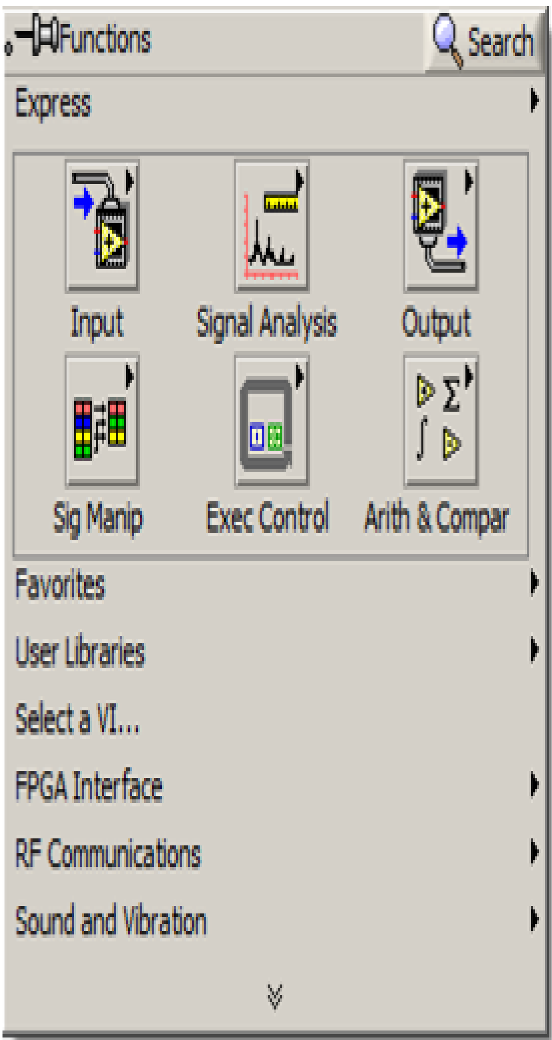
A VI icon is a graphical representation of a VI. It appears in the top right corner of a block diagram or a front panel window. When a VI is inserted into a block diagram as a subVI, its icon is displayed.

A connector pane defines VI inputs (controls) and outputs (indicators). One can change the number of inputs and outputs by using different connector pane patterns. In [link], a VI icon is shown at the top right corner of the block diagram, and its corresponding connector pane, with two inputs and one output, is shown at the top right corner of the front panel.

Graphical Environment

Functions Palette

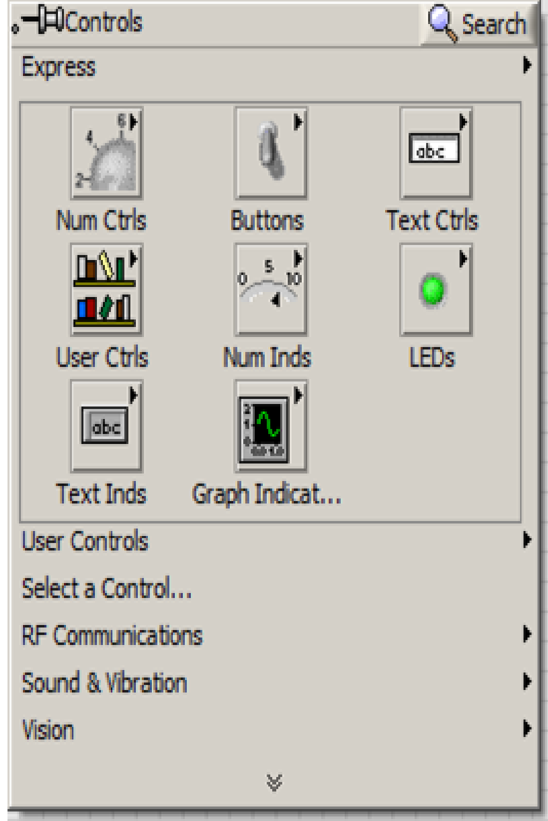
The Functions palette (see [link]) provides various function VIs or blocks to build a system. View this palette by right-clicking on an open area of a block diagram. Note that this palette can be displayed only in a block diagram.



Functions Palette

Controls Palette

The Controls palette (see [link]) features front panel controls and indicators. View this palette by right-clicking on an open area of a front panel. Note that this palette can be displayed only in a front panel.



Controls Palette

Tools Palette

The Tools palette offers various mouse cursor operation modes for building or debugging a VI. The Tools palette and the frequently used tools are shown in [link].

Icon	Tool
	Automatic Tool Selection
	Operating tool
	Positioning tool
	Labeling tool
	Wiring tool
	Probe tool

Tools Palette

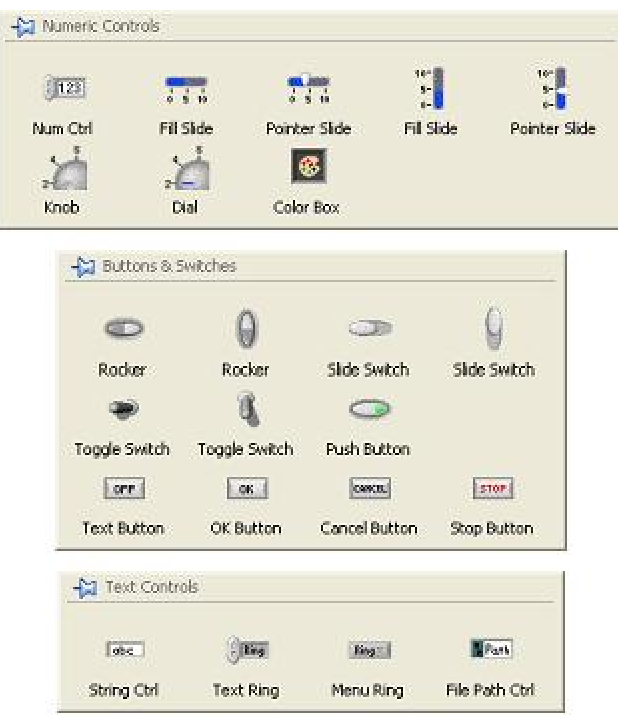
Each tool is used for a specific task. For example, use the wiring tool to wire objects in a block diagram. If one enables the automatic tool selection mode by clicking on the **Automatic Tool Selection** button, LabVIEW selects the best matching tool based on a current cursor position.

Building a Front Panel

In general, one constructs a VI by going back and forth between a front panel and block diagram, placing inputs/outputs on the front panel and building blocks on the block diagram.

Controls

Controls make up the inputs to a VI. Controls grouped in the **Numeric Controls palette**(Controls → Express → Num Ctrls) are used for numerical inputs, controls grouped in the **Buttons & Switches palette(Controls → Express → Buttons)** are used for Boolean inputs, and controls grouped in the **Text Controls palette(Controls →Express →Text Ctrls)** are used for text and enumeration inputs. These control options are displayed in [link].



Control Palettes

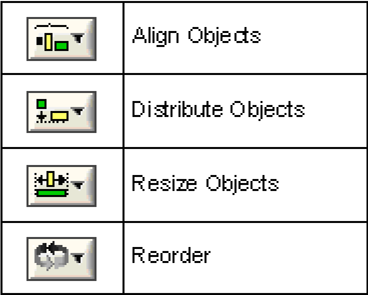
Indicators

Indicators make up the outputs of a VI. Indicators grouped in the **Numeric Indicators palette(Controls→ Express → Numeric Inds)** are used for numerical outputs, indicators grouped in the **LEDs palette(Controls → Express → LEDs)** are used for Boolean outputs, indicators grouped in the **Text Indicators palette(Controls → Express → Text Inds)** are used for text outputs, and indicators grouped in the **Graph Indicators palette(Controls → Express → Graph Indicators)** are used for graphical outputs. These indicator options are displayed in [link].

Indicator Palettes

Align, Distribute and Resize Objects

The menu items on the front panel toolbar (see [link]) provide options to align and orderly distribute objects on the front panel. Normally, after one places controls and indicators on a front panel, these options can be used to tidy up their appearance.



	Align Objects
	Distribute Objects
	Resize Objects
	Reorder

Menu to Align, Distribute, Resize and Reorder Objects

Building a Block Diagram

Express VI and Function

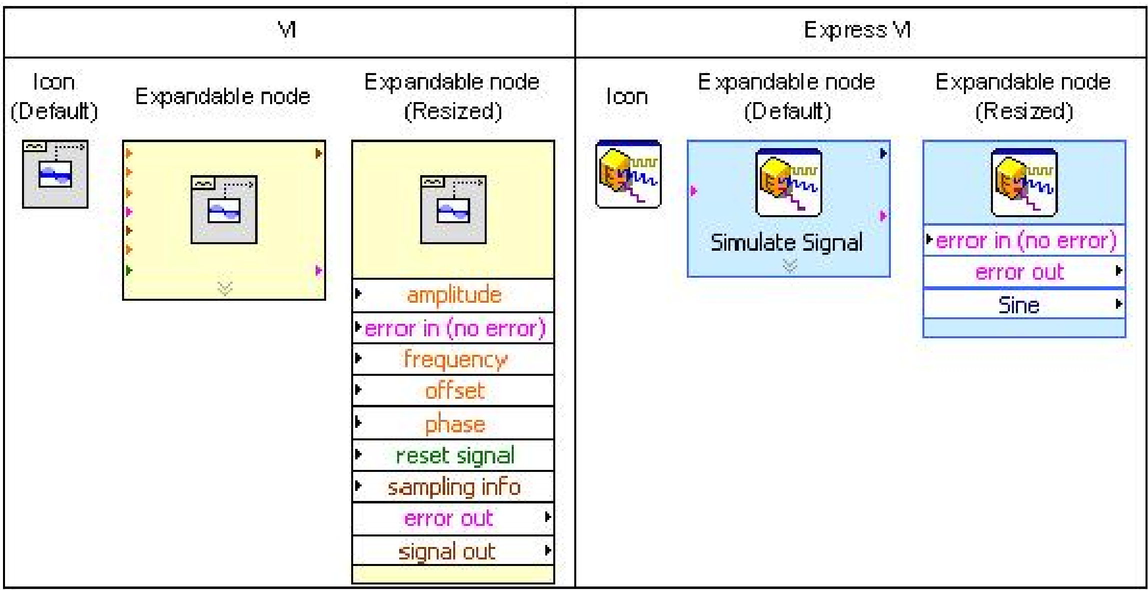
Express VIs denote higher-level VIs configured to incorporate lower-level VIs or functions. These VIs are displayed as expandable nodes with a blue background. Placing an Express VI in a block diagram opens a configuration dialog window to adjust the Express VI parameters. As a result, Express VIs demand less wiring. The configuration window can be opened by double-clicking on its Express VI.

Basic operations such as addition or subtraction are represented by functions. [link] shows three examples corresponding to three block diagram objects (VI, Express VI and function).

Block Diagram Objects: (a) VI, (b) Express VI, (c) Function

One can display subVIs or Express VIs as icons or expandable nodes. If a subVI is displayed as an expandable node, the background appears yellow. Icons can be used to save space in a block diagram and expandable nodes

can be used to achieve easier wiring or better readability. One can resize expandable nodes to show their connection nodes more clearly. Three appearances of a VI/Express VI are shown in [link].



Icon versus Expandable Node

Terminal Icons

Front panel objects are displayed as terminal icons in a block diagram. A terminal icon exhibits an input or output as well as its data type. [link] shows two terminal icon examples consisting of a double precision numerical control and indicator. As shown in this figure, one can display terminal icons as data type terminal icons to conserve space in a block diagram.

	Control	Indicator
Terminal Icons	1.23 DBL	1.23 DBL
Data Type Terminal Icons	DBL	DBL

Terminal Icon Examples
Displayed in a Block Diagram

Wires

Wires transfer data from one node to another in a block diagram. Based on the data type of a data source, the color and thickness of its connecting wires change.

Wires for the basic data types used in LabVIEW are shown in [link]. In addition to the data types shown in this figure, there are some other specific data types. For example, the dynamic data type is always used for Express VIs, and the waveform data type, which corresponds to the output from a waveform generation VI, is a special cluster of waveform components incorporating trigger time, time interval and data value.

Wire Type	Scalar	1D Array	2D Array	Color
Numeric				Orange (Floating point) Blue (Integer)
Boolean				Green
String				Pink

Basic Wire Types

Structures

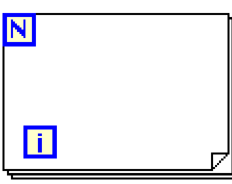
A structure is represented by a graphical enclosure. The graphical code enclosed in the structure gets repeated or executed conditionally. A loop structure is equivalent to a for loop or a while loop statement in text-based programming languages, while a case structure is equivalent to an if-else statement.

For Loop

A for loop structure is used to perform repetitions. As illustrated in [link], the displayed border indicates a **for loop** structure, where the count terminal

represents the number of times the loop is to be repeated. It is set by wiring a value from outside of the loop to it. The iteration terminal

denotes the number of completed iterations, which always starts at zero.



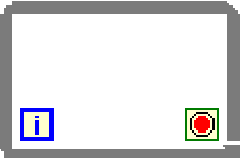
For Loop

While Loop

A **while loop** structure allows repetitions depending on a condition (see [link]). The conditional terminal

initiates a stop if the condition is true. Similar to a **for loop**, the iteration terminal

provides the number of completed iterations, always starting at zero.



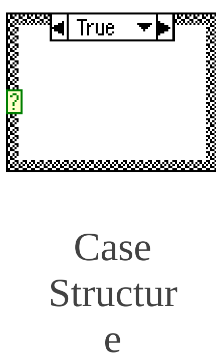
While Loop

Case Structure

A **case structure** (see [link]) allows the running of different sets of operations depending on the value it receives through its selector terminal, which is indicated by

. In addition to Boolean type, the input to a selector terminal can be of integer, string, or enumerated type. This input determines which case to execute. The case selector

shows the status being executed. Cases can be added or deleted as needed.



Case Structure

Grouping Data: Array and Cluster

An array represents a group of elements having the same data type. An array consists of data elements having a dimension up to $2^{31} - 1$. For example, if a random number is generated in a loop, it is appropriate to build the output as an array because the length of the data element is fixed at 1 and the data type is not changed during iterations.

Similar to the structure data type in text-based programming languages, a cluster consists of a collection of different data type elements. With clusters, one can reduce the number of wires on a block diagram by bundling different data type elements together and passing them to only one terminal. One can add or extract an individual element to or from a cluster by using the cluster functions such as **Bundle by Name** and **Unbundle by Name**.

Debugging and Profiling VIs

Probe Tool

VIs can be debugged as they run by checking values on wires with the Probe tool. Note that the Probe tool can be accessed only in a block diagram window.

With the Probe tool, breakpoints and execution highlighting, one can identify the source of an incorrect or an unexpected outcome. To visualize the flow of data during program execution, a breakpoint can be used to pause the execution of a VI at a specific location.

Profile Tool

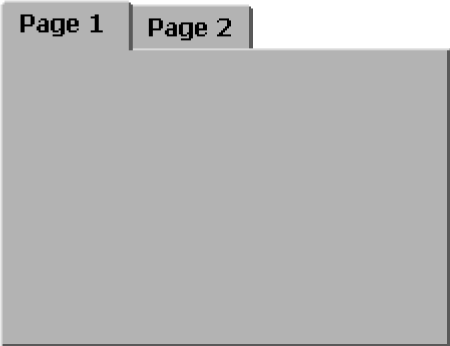
Timing and memory usage information – in other words, how long a VI takes to run and how much memory it consumes – can be gathered with the Profile tool. It is required to make sure that a VI is stopped before setting up a Profile window.

An effective way to become familiar with LabVIEW programming is to review examples. In the lab that follows, we explore most of the key LabVIEW programming features by building simple VIs.

Containers and Decoration Tools

Containers and Decoration tools can be used to organize front panel controls and indicators. Container tools are grouped in the **Containers pallete(Controls → Modern → Containers → Classic → Classic Containers)** and Decoration tools are grouped in the **Decorations pallete(Controls → Modern → Decorations)**.

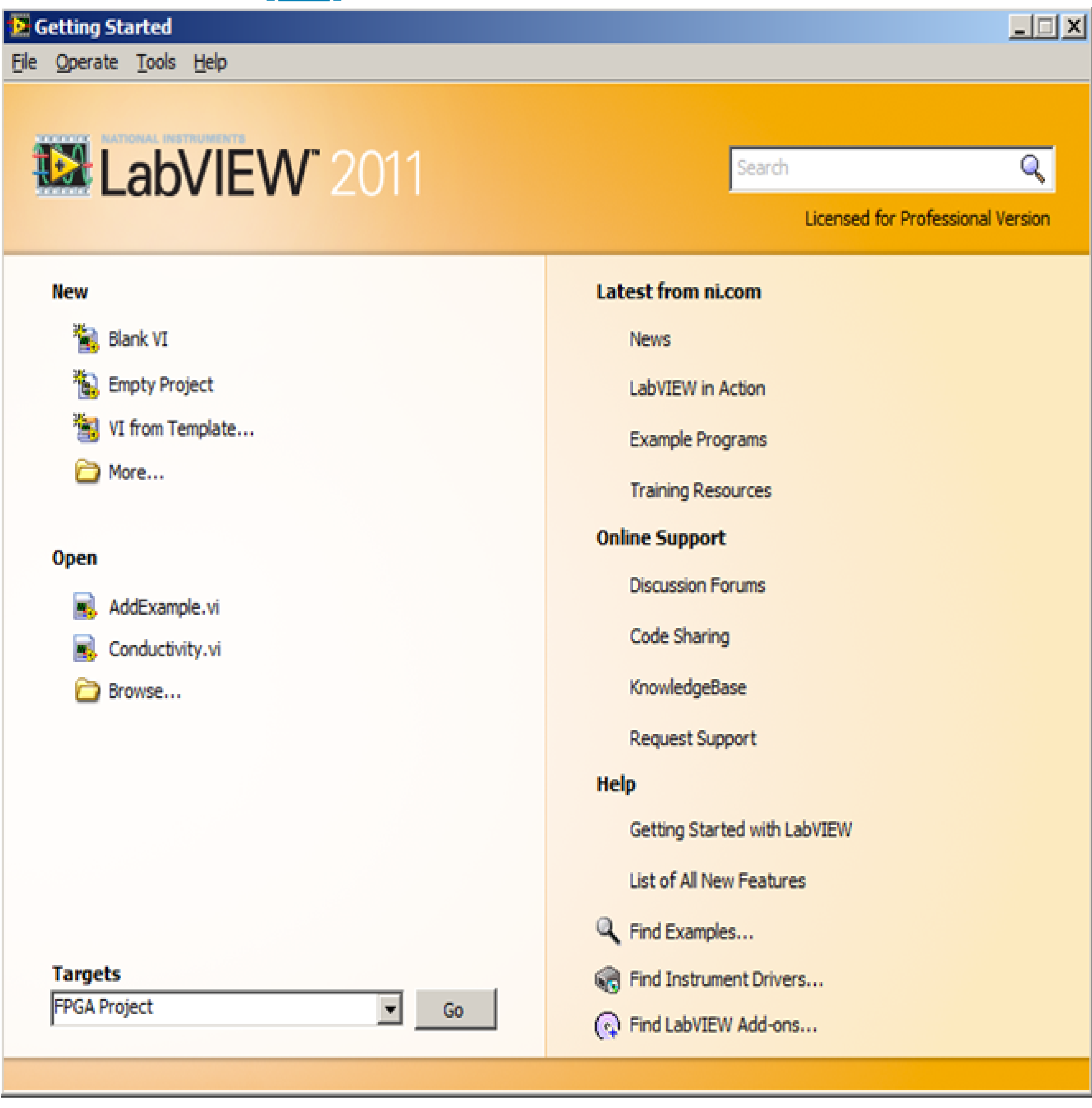
One can use **Tab Control(Controls → Modern → Containers → Tab Control → Classic → Classic Containers →Tab Control)** to display various controls and indicators within a limited screen area. This feature helps one to organize controls and indicators under different tabs as illustrated in [link]. To add more tabs or delete tabs, right-click the border area and choose one of the following options: **Add Page After**, **Add Page Before**, **Duplicate Page** or **Remove Page**.



Tab Control

Lab 1: Introduction to LabVIEW

The objective of this lab is to offer an initial hands-on experience in building a VI. More detailed explanations of the LabVIEW features mentioned here can be found in the [link], [link], and [link]. One can launch LabVIEW 2011 (the latest version at the time of this publication) by double-clicking on the LabVIEW 2011 icon, which opens the dialog window shown in [link].



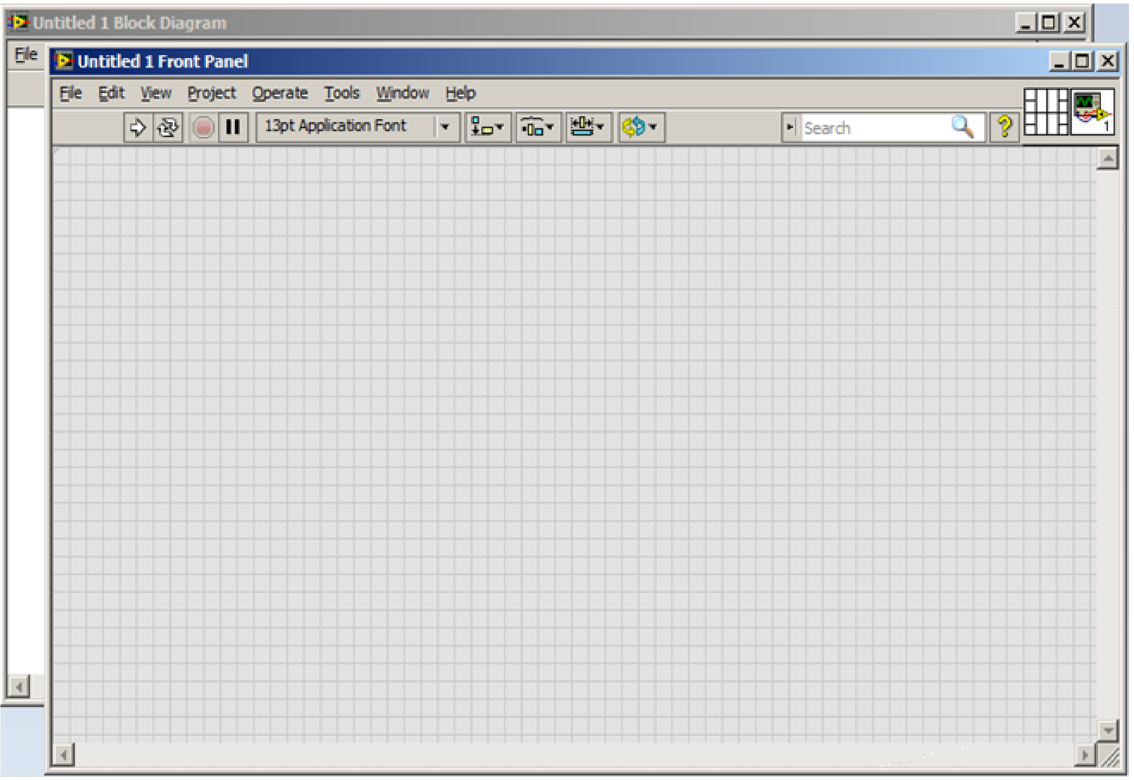
Starting LabVIEW

Building a Simple VI

To become familiar with the LabVIEW programming environment, let us calculate the sum and average of two input values in the following step-by-step example.

Sum and Average VI Example Using Graphical Programming

To create a new VI, click on the Blank VI under New, as shown in [link]. This can also be done by choosing **File → New VI** from the menu. As a result, a blank front panel and a blank block diagram window appear, see [link]. Remember that a front panel and block diagram coexist when one builds a VI, meaning that every VI will have both a front panel and an associated block diagram.



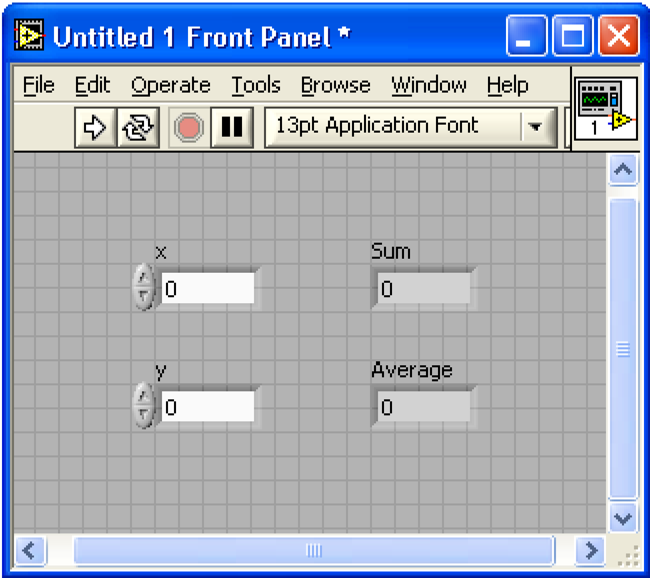
Blank VI

The number of VI inputs and outputs is dependent on the VI function. In this example, two inputs and two outputs are needed, one output generating the sum and the other generating the average of two input values. Create the inputs by locating two numeric controls on the front panel. This can be done by right-clicking on an open area of the front panel to bring up the **Controls palette**, followed by choosing **Controls** → **Modern** → **Numeric** → **Numeric Control**. Each numeric control automatically places a corresponding terminal icon on the block diagram. Double-clicking on a numeric control highlights its counterpart on the block diagram and vice versa.

Next, label the two inputs as x and y using the Labeling tool from the **Tools Palette**, which can be displayed by choosing **View** → **Tools Palette** from the menu bar. Choose the Labeling tool and click on the default labels, **Numeric** and **Numeric 2**, to edit them. Alternatively, if the automatic tool selection mode is enabled by clicking **Automatic Tool Selection** in the **Tools Palette**, the labels can be edited by simply double-clicking on the default labels. Editing a label on the front panel changes its corresponding terminal icon label on the block diagram and vice versa.

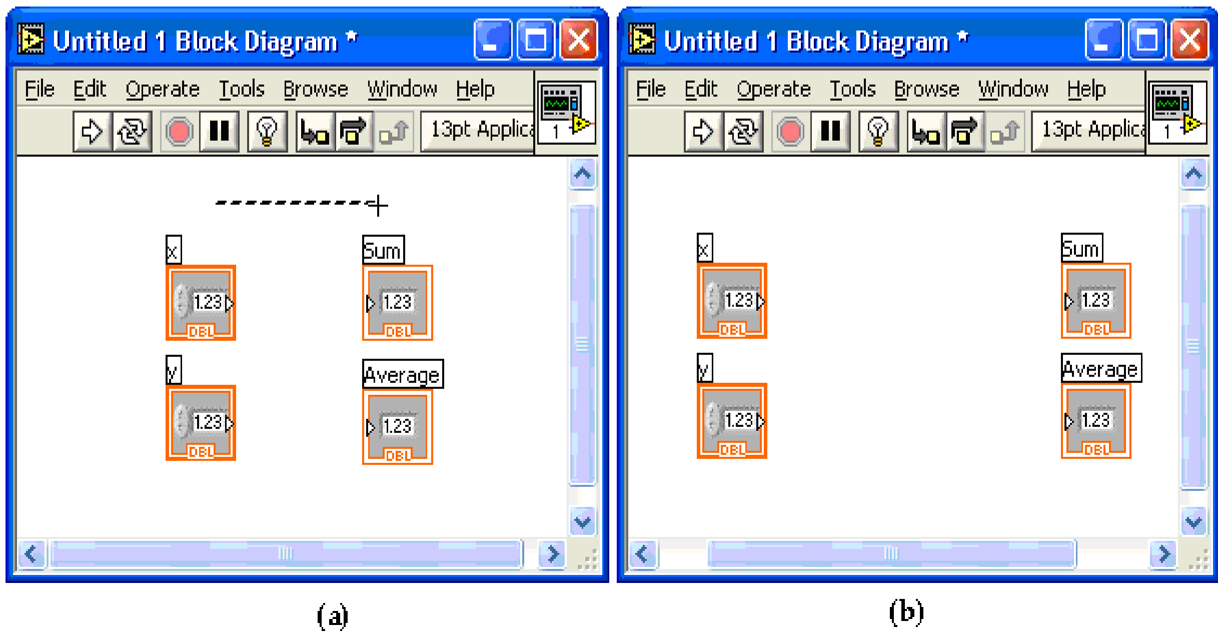
Similarly, the outputs are created by locating two numeric indicators **(Controls** → **Modern** → **Numeric** →**Numeric Indicator)** on the front panel. Each numeric indicator automatically places a corresponding terminal icon on the block diagram. Edit the labels of the indicators to read “Sum“ and “Average.”

For a better visual appearance, one can align, distribute and resize objects on a front panel window using the front panel toolbar. To do this, select the objects to be aligned or distributed and apply the appropriate option from the toolbar menu. [link] shows the configuration of the front panel just created.



Front Panel Configuration

Now build a graphical code on the block diagram to perform the summation and averaging operations. Note that <Ctrl-E> toggles between a front panel and a block diagram window. If objects on a block diagram are too close to insert other functions or VIs in-between, one can insert a horizontal or vertical space by holding down the <Ctrl> key to create space horizontally and/or vertically. As an example, [link]b illustrates a horizontal space inserted between the objects shown in [link]a.

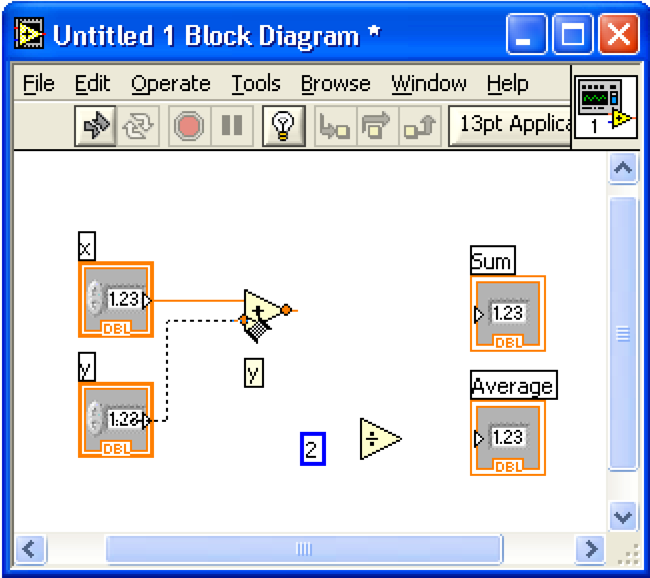


Inserting Horizontal/Vertical Space: (a) Creating Space While Holding Down the <Ctrl> Key, (b) Inserted Horizontal Space.

Next, place an **Add** function **(Functions →Express →Arith & Compar →Express Numeric →Add)** and a **Divide** function **(Functions →Express →Arith & Comp →Express Numeric →Divide)** on the block diagram. Enter the divisor, in this case 2, in a **Numeric Constant(Functions →Express →Arith & Compar →Express Numeric →Numeric Constant)** and connect it to the y terminal of the **Divide** function using the Wiring tool.

To achieve proper data flow, wire functions, structures and terminal icons on a block diagram using the Wiring tool. To wire these objects, point the Wiring tool at the terminal of the function or subVI to be wired, left-click on the terminal, drag the mouse to a destination terminal and left-click once again. [link] illustrates the wires placed between the terminals of the numeric controls and the input terminals of the **Add** function. Notice that the label of a terminal gets displayed whenever one moves the cursor over the terminal if the automatic tool selection mode is enabled. Also, note that the **Run** button

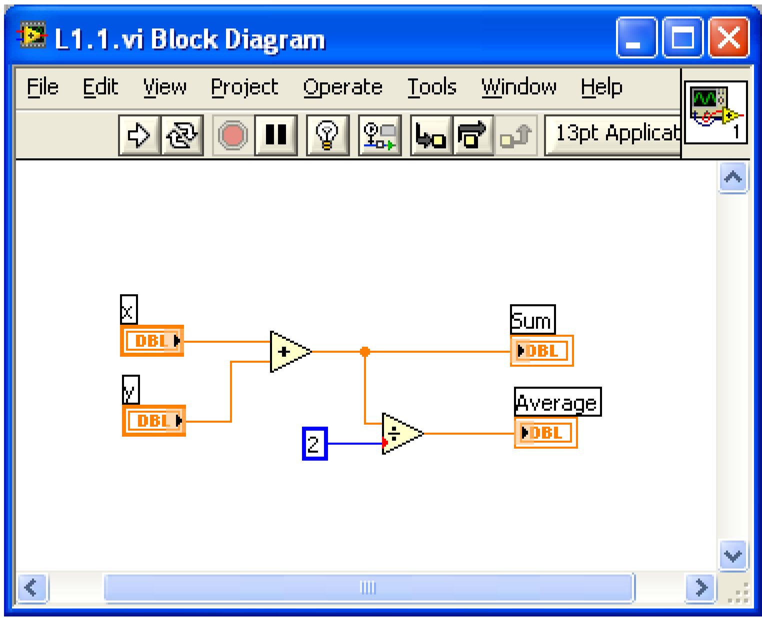
on the toolbar remains broken until one completes the wiring process.



Wiring Block Diagram Objects.

For better block diagram readability, one can clean up wires hidden behind objects or crossed over other wires by right-clicking on them and choosing **Clean Up Wire** from the shortcut menu. Any broken wires can be cleared by pressing <Ctrl-B> or **Edit** →**Remove Broken Wires**.

To view or hide the label of a block diagram object, such as a function, right-click on the object and check (or uncheck) **Visible Items** →**Label** from the shortcut menu. Also, one can show a terminal icon corresponding to a numeric control or indicator as a data type terminal icon by right-clicking on the terminal icon and unchecking **View As Icon** from the shortcut menu. [link] shows an example where the numeric controls and indicators are depicted as data type terminal icons. The notation DBL indicates double precision data type.



Completed Block Diagram.

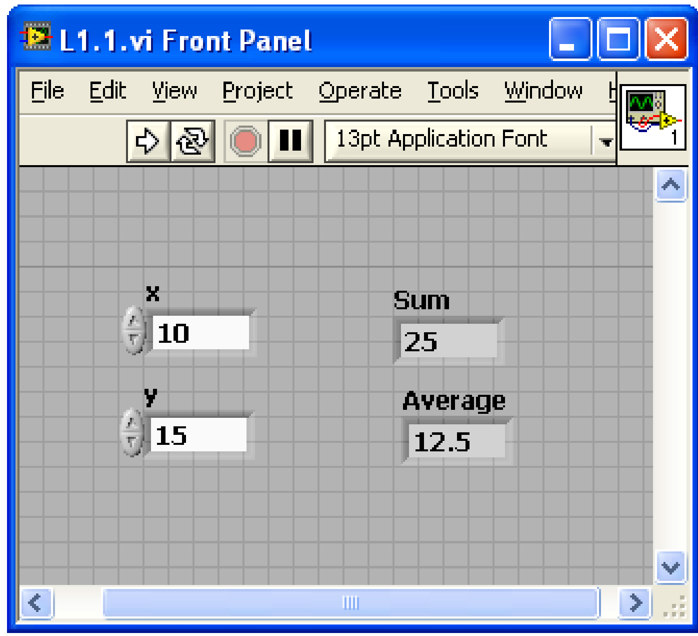
It is worth noting that there is a shortcut to build the above VI. Instead of choosing the numeric controls, indicators or constants from the Controls or Functions palette, one can use the shortcut menu **Create**, activated by right-clicking on a terminal of a block diagram object such as a function or a subVI. As an example of this approach, create a blank VI and locate an **Add** function. Right-click on its x terminal and choose **Create** →**Control** from the shortcut menu to create and wire a numeric control or input. This locates a numeric control on the front panel as well as a corresponding terminal icon on the block diagram. The label is automatically set to x. Create a second numeric control by right-clicking on the y terminal of the **Add** function. Next, right-click on the output terminal of the **Add** function and choose **Create** →**Indicator** from the shortcut menu. A data type terminal icon, labeled as x+y, is created on the block diagram as well as a corresponding numeric indicator on the front panel.

Next, right-click on the y terminal of the **Divide** function to choose **Create** →**Constant** from the shortcut menu. This creates a numeric constant as the divisor and wires its y terminal. Type the value 2 in the numeric constant. Right-click on the output terminal of the **Divide** function, labeled as x/y,

and choose **Create →Indicator** from the shortcut menu. If the wrong option is chosen, the terminal does not get wired. An incorrect terminal option can easily be changed by right-clicking on the terminal and choosing **Change to Control** from the shortcut menu.

To save the created VI for later use, choose **File →Save** from the menu or press <Ctrl-S> to bring up a dialog window to enter a name. Type “Sum and Average” as the VI name and click **Save**.

To test the functionality of the VI, enter some sample values in the numeric controls on the front panel and run the VI by choosing **Operate →Run**, by pressing <Ctrl-R> or by clicking the **Run** button on the toolbar. From the displayed output values in the numeric indicators, the functionality of the VI can be verified. [link] illustrates the outcome after running the VI with two inputs, 10 and 15.



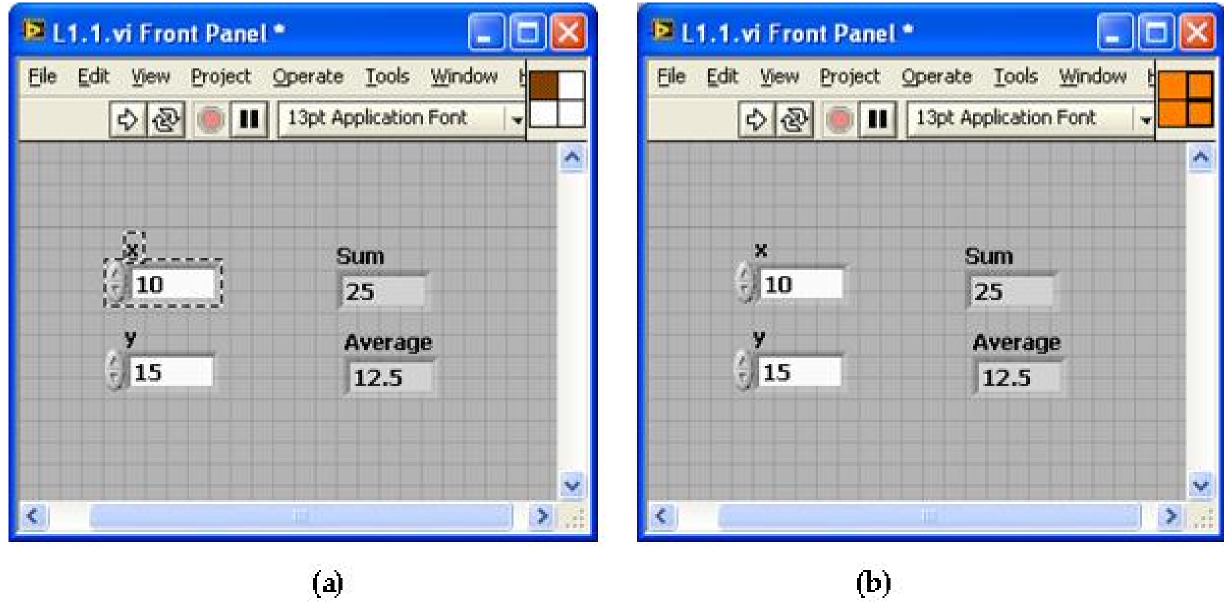
VI Verification

SubVI Creation

If it is desired to use a VI as part of a higher-level VI, one needs to configure its connector pane. A connector pane assigns inputs and outputs of a subVI to its terminals through which data are exchanged.

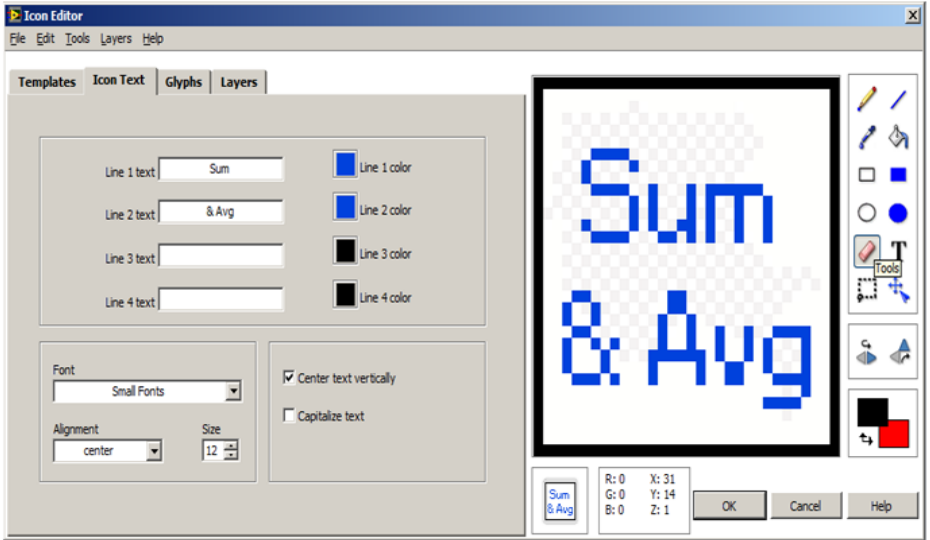
The default pattern of a connector pane is determined based on the number of controls and indicators. In general, the terminals on the left side of a connector pane pattern are used for inputs and the ones on the right side for outputs. One can add terminals to or remove them from a connector pane by right-clicking and choosing **Add Terminal** or **Remove Terminal** from the shortcut menu. If the number of inputs/outputs or the distribution of terminals are changed, the connector pane pattern can be replaced with a new one by right-clicking and choosing **Patterns** from the shortcut menu. Once a pattern is selected, one needs to reassign each terminal to a control or an indicator by using the Wiring tool or by enabling the automatic tool selection mode.

[link]a illustrates how to assign a Sum and Average VI terminal to a numeric control. The completed connector pane is shown in [link]b. Notice that the output terminals have thicker borders. The color of a terminal reflects its data type.



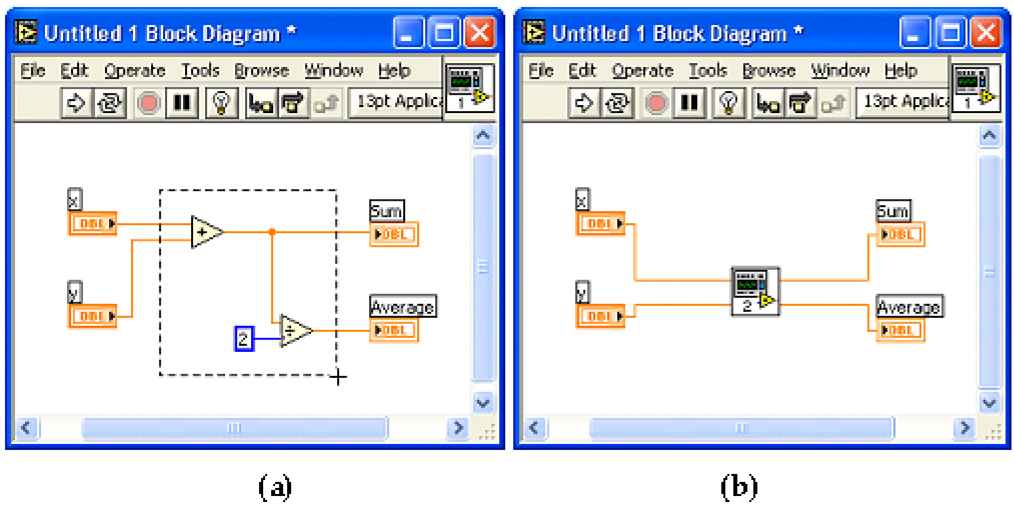
Connector Pane: (a) Assigning a Terminal to a Control, (b) Completed Terminal Assignment.

Considering that a subVI icon is displayed on the block diagram of a higher-level VI, it is important to edit the subVI icon for it to be explicitly identifiable. Double-clicking on the top-right corner icon of a block diagram opens the Icon Editor. The Icon Editor tools are similar to those in other graphical editors, such as Microsoft Paint. Editing the Sum and Average VI icon is illustrated in [link].



Editing SubVI Icon.

A subVI can also be created from a section of a VI. To do so, select the nodes on the block diagram to be included in the subVI, as shown in [link]a. Then, choose **Edit** →**Create SubVI** to insert a new subVI icon. [link]b illustrates the block diagram with an inserted subVI. One can open and edit this subVI by double-clicking on its icon on the block diagram. Save this subVI as **Sum and Average.vi**. This subVI performs the same function as the original Sum and Average VI.



Creating a SubVI: (a) Selecting Nodes to Make a SubVI, (b) Inserted SubVI Icon.

Using Structures and SubVIs

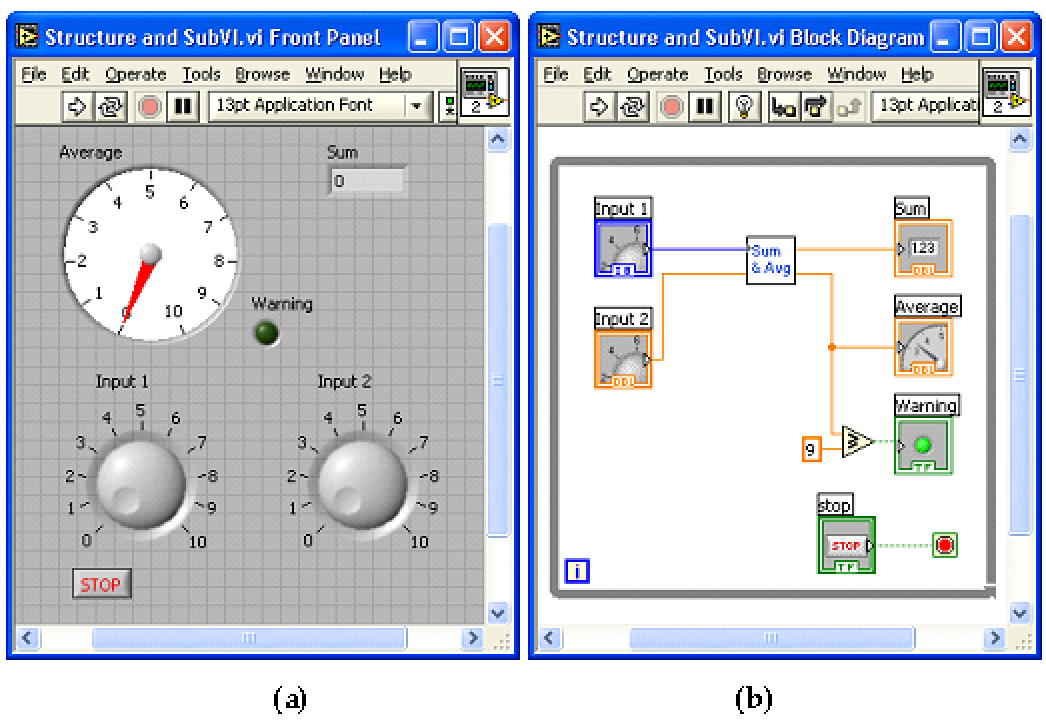
Now let us consider another example to understand the use of structures and subVIs. In this example, we use a VI to show the sum and average of two input values, which are altered in a continuous fashion. If the average of the two inputs becomes greater than a preset threshold value, a LED warning light turns on.

First, build a front panel as shown in [link]a. For the inputs, consider two **Knobs(Controls →Modern →Numeric →Knob)**. Adjust the size of the knobs by using the Positioning tool. One can modify knob properties such as precision and data type by right-clicking and choosing **Properties** from the shortcut menu. A Knob Properties dialog box opens and an **Appearance** tab is shown by default. Edit the label of one of the knobs to read Input 1. Select the **Data Type** tab, click **Representation** and select **Byte** to change the data type from double precision to byte. One can also perform this by right-clicking on the knob and choosing **Representation →Byte** from the shortcut menu. In the **Data Type** tab, a default value needs

to be specified. In this example, the default value is considered to be 0. The default value can be set by right-clicking on the control and choosing **Data Operations →Make Current Value Default** from the shortcut menu. Also, this control can be set to a default value by right-clicking and choosing **Data Operations →Reinitialize to Default Value** from the shortcut menu.

Label the second knob as Input 2 and repeat all the adjustments as carried out for the first knob except for the data representation part. Specify the data type of the second knob to be double precision to demonstrate the difference in the outcome. As the final front panel configuration step, align and distribute the objects using the appropriate buttons on the front panel toolbar.

To set the outputs, locate and place a numeric indicator, a round LED **(Controls →Modern →Boolean →Round LED)** and a gauge **(Controls →Modern →Numeric →Gauge)**. Edit the labels of the indicators as shown in [link].

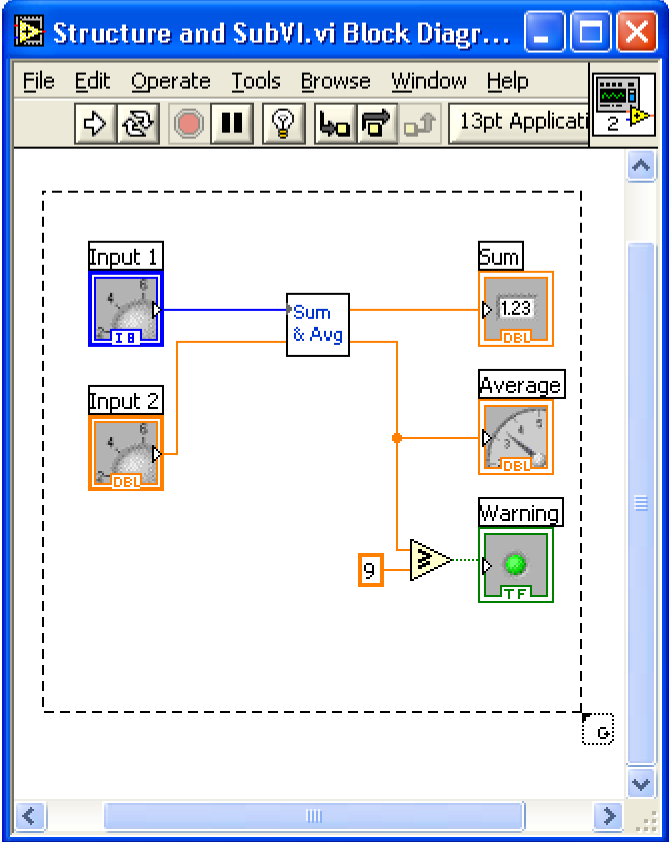


Example of Structure and SubVI: (a) Front Panel, (b) Block

Diagram.

Locate a **Greater or Equal?** function from **Functions** →**Programming** →**Comparison** →**Greater or Equal?** to compare the average output of the subVI with a threshold value. Create a wire branch on the wire between the Average terminal of the subVI and its indicator via the Wiring tool. Then, extend this wire to the x terminal of the Greater or Equal? function. Right-click on the y terminal of the Greater or Equal? function and choose **Create** →**Constant** to place a numeric constant. Enter 9 in the numeric constant and wire the round LED, labeled as Warning, to the x>=y? terminal of this function to provide a Boolean value.

To run the VI continuously, use a while loop structure. Choose **Functions** →**Programming** →**Structures** →**While Loop**to create a **while loop**. Change the size by dragging the mouse to enclose the objects in the **while loop**, as illustrated in [link].



While Loop Enclosure.

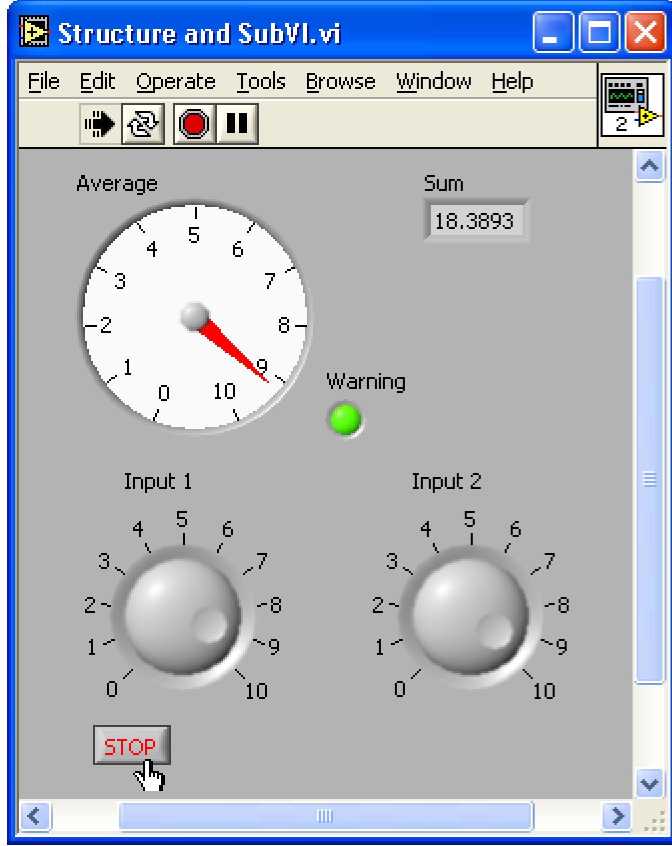
Once this structure is created, its boundary, together with the loop iteration terminal

and conditional terminal

, get shown on the block diagram. If one creates the **while loop** by using **Functions** →**Programming** →**Structures** → **While Loop**, the **Stop** button is not included as part of the structure. One can create this button by right-clicking on the conditional terminal and choosing **Create** →**Control** from the shortcut menu. It is possible to wire a Boolean condition to a

conditional terminal, instead of a **Stop** button, to stop the loop programmatically.

Next run the VI to verify its functionality. After clicking the Run button on the toolbar, adjust the knobs to alter the inputs. Verify whether the average and sum are displayed correctly in the gauge and numeric indicators. Note that only integer values can be entered via the Input 1 knob while real values can be entered via the Input 2 knob. This is due to the data types associated with these knobs. The Input 1 knob is set to byte type, in other words, I8 or 8-bit signed integer. As a result, one can enter only integer values within the range -128 and 127. Considering that the minimum and maximum values of this knob are set to 0 and 10, respectively, one can enter only integer values from 0 to 10 for this input.

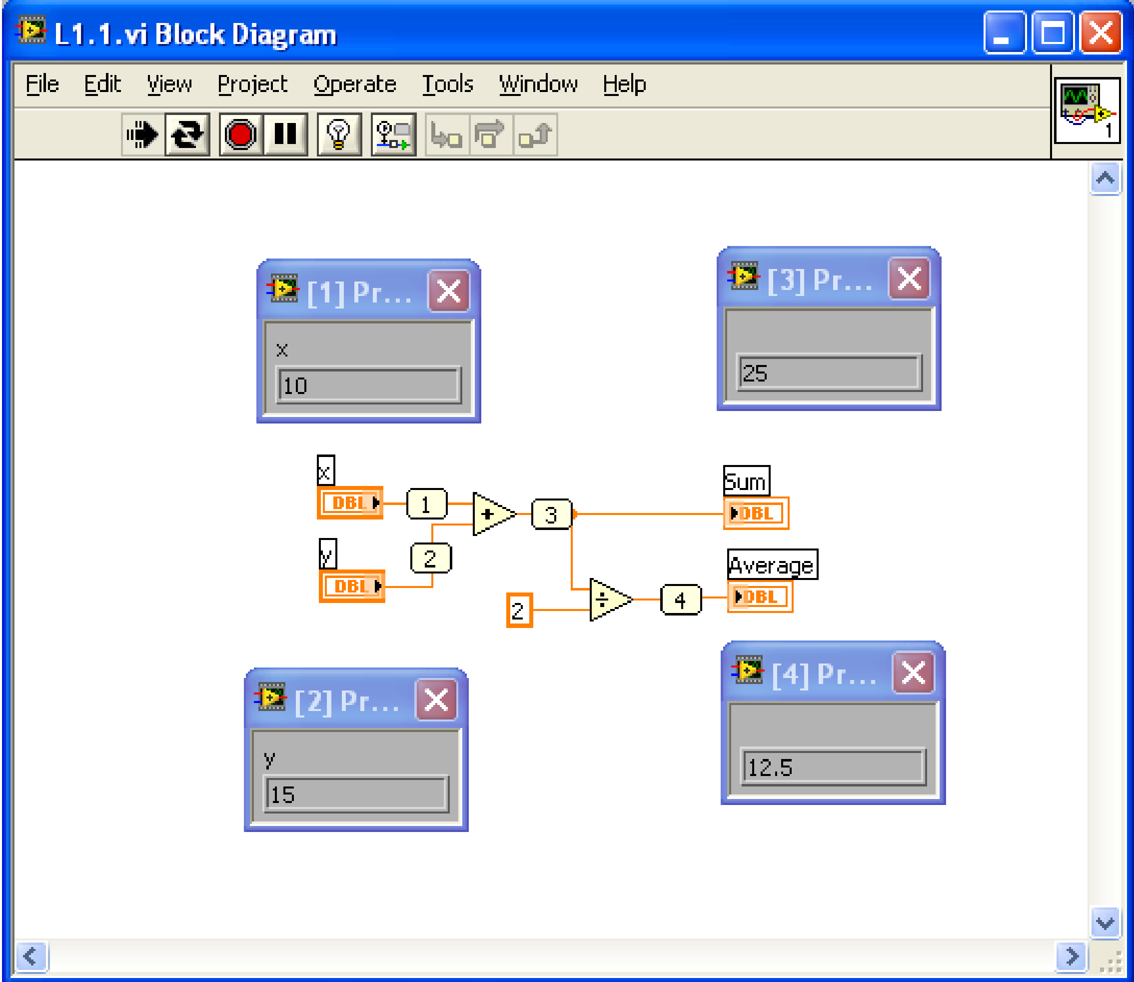


Front Panel as VI Runs.

Debugging VIs: Probe Tool

Use the Probe tool to observe data that are being passed while a VI is running. A probe can be placed on a wire by using the Probe tool or by right-clicking on a wire and choosing **Probe** from the shortcut menu. Probes can also be placed while a VI is running.

Placing probes on wires creates probe windows through which one can observe intermediate values. As an example of using custom probes, use four probe windows at the probe locations 1 through 4 in the Sum and Average VI to probe the values at those locations. These probes and their locations are illustrated in [link].



Probe Tool.

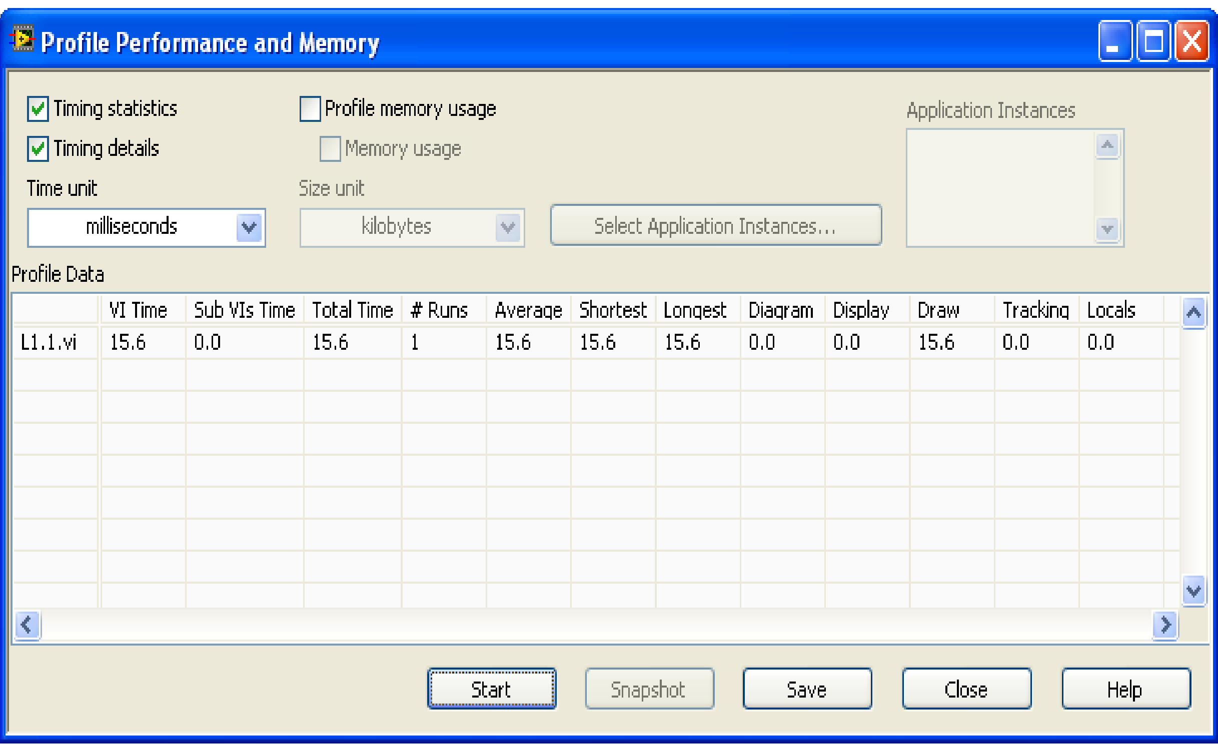
Profile Tool

With the Profile tool, one can gather timing and memory usage information. Make sure to stop the VI before selecting **Tools** →**Profile** →**Performance and Memory** to open a Profile window.

Place a checkmark in the **Timing Statistics** checkbox to display timing statistics of the VI. The **Timing Details** option offers more detailed VI statistics such as drawing time. To profile memory usage as well as timing, check the **Memory Usage** checkbox after checking the **Profile Memory**

Usage checkbox. Note that this option can slow down VI execution. Start profiling by clicking the **Start** button on the profiler, then run the VI. Obtain a snapshot of the profiler information by clicking on the **Snapshot** button. After viewing the timing information, click the **Stop** button. The profile statistics can be stored in a text file by clicking the **Save** button.

An outcome of the profiler is shown in [link] after running the Sum and Average or L1.1 VI. [link] provides more details on the Profile tool.



Profile Window after Running Sum and Average VI.

Lab Exercises

Exercise:

Problem:

Build a VI to compute the variance of an array x. The variance σ is defined as:

Equation:

$$\sigma = \frac{1}{N}\sum_{j=1}^{N}(x_j - \mu)^2$$

where μdenotes the average of the array x. For x, use all the integers from 1 to 1000.

Solution:

Insert Solution Text Here

Exercise:

Problem:

Build a VI to check whether a given positive integer n is a prime number and display a warning message if it is not a prime number.

Solution:

Insert Solution Text Here

Exercise:

Problem:

Build a VI to generate the first Nprime numbers and store them using an indexing array. Display the outcome.

Solution:

Insert Solution Text Here

Exercise:

Problem:

Build a VI to sort N integer numbers (positive or negative) in ascending or descending order.

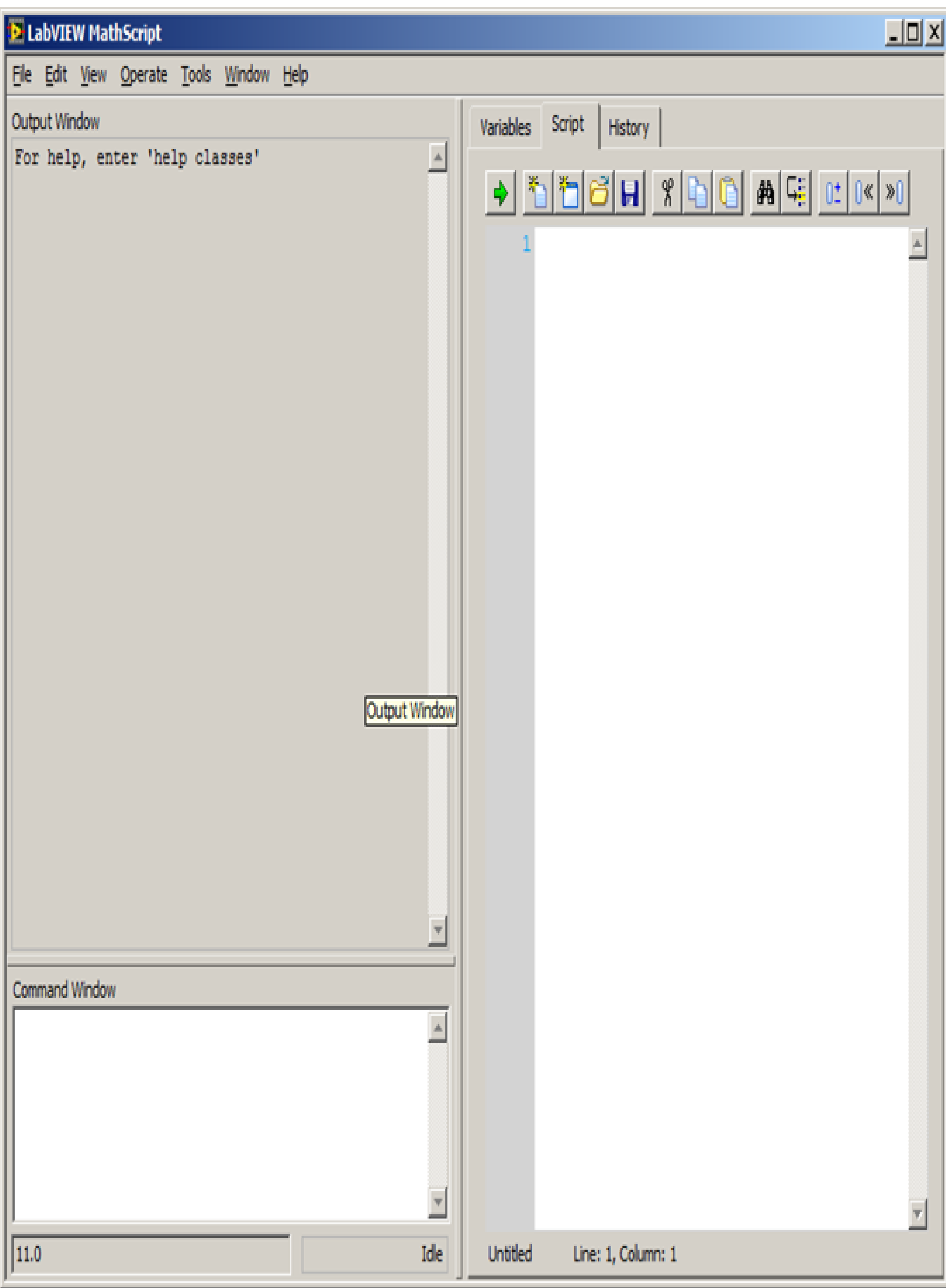
Solution:

Insert Solution Text Here

LabVIEW MathScript and Hybrid Programming

In signals and systems lab courses, .m file coding is widely used. LabVIEW MathScript is a feature of the newer versions of LabVIEW that allows one to include .m files within its graphical environment. As a result, one can perform hybrid programming, that is, a combination of textual and graphical programming, when using this feature. This chapter provides an introduction to MathScript or .m file textual coding. See [link] and [link] for advanced MathScript aspects.

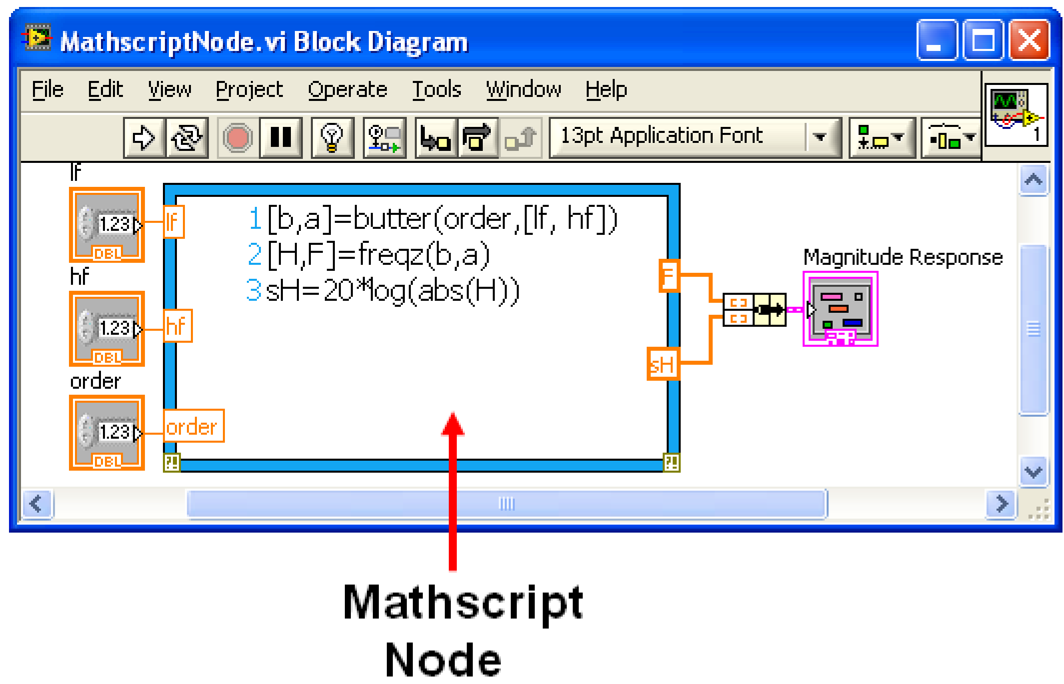
MathScripting can be done via the LabVIEW MathScript interactive window or node. The LabVIEW MathScript interactive window, shown in [link], consists of a Command Window, an Output Window and a MathScript Window. The Command Window interface allows one to enter commands and debug script or to view help statements for built-in functions. The Output Window is used to view output values and the MathScript Window interface to display variables and command history as well as edit scripts. With script editing, one can execute a group of commands or textual statements.



LabVIEW MathScript
File Edit View Operate Tools Window Help
Output Window
For help, enter 'help classes'
Output Window
Variables
Script
History
1
Command Window
11.0
Idle
Untitled
Line: 1, Column: 1

LabVIEW MathScript Interactive Window

A LabVIEW MathScript node represents the textual .m file code via a blue rectangle as shown in [link]. Its inputs and outputs are defined on the border of this rectangle for transferring data between the graphical environment and the textual code. For example, as indicated in [link], the input variables on the left side, namely lf, hf and order, transfer values to the .m file script, and the output variables on the right side, F and sH, transfer values to the graphical environment. This process allows .m file script variables to be used within the LabVIEW graphical programming environment.



LabVIEW MathScript Node Interface

Lab 2: LabVIEW MathScript and Hybrid Programming

Arithmetic Operations

There are four basic arithmetic operators in .m files:

`+ addition`

`- subtraction`

`* multiplication`

`/ division (for matrices, it also means inversion)`

The following three operators work on an element-by-element basis:

`.* multiplication of two vectors, element-wise`

`./ division of two vectors, element-wise`

`.^ raising all the elements of a vector to a power`

As an example, to evaluate the expression $a^3 + \sqrt{\mathrm{bd}} - 4\mathrm{c}$, where $a = 1.2$, $b = 2.3$, $c = 4.5$and $d = 4$, type the following commands in the **Command Window** to get the answer (ans) :

```
>> a=1.2;
>> b=2.3;
>> c=4.5;
>> d=4;
>> a^3+sqrt(b*d)-4*c
ans =
```

`-13.2388`

Note the semicolon after each variable assignment. If the semicolon is omitted, the interpreter echoes back the variable value.

Vector Operations

Consider the vectors $\mathbf{x} = [x_1, x_2, \ldots, x_n]$and $\mathbf{y} = [y_1, y_2, \ldots, y_n]$. The following operations indicate the resulting vectors:

x*.y $= [x_1 y_1, x_2 y_2, \ldots, x_n y_n]$

x./y $= \left[\frac{x_1}{y_1}, \frac{x_2}{y_3}, \ldots, \frac{x_n}{y_n}\right]$

x.^p $= [x_1^p, x_2^p, \ldots, x_n^p]$

Note that because the boldfacing of vectors/matrices are not used in .m files, in the notation adopted in this book, no boldfacing of vectors/matrices is shown to retain consistency with .m files.

The arithmetic operators + and – can be used to add or subtract matrices, vectors or scalars. Vectors denote one-dimensional arrays and matrices denote multidimensional arrays. For example,

```
>> x=[1,3,4]

>> y=[4,5,6]

>> x+y

ans=

5 8 10
```

In this example, the operator + adds the elements of the vectors x and y, element by element, assuming that the two vectors have the same dimension, in this case 1×3 or one row with three columns. An error

occurs if one attempts to add vectors having different dimensions. The same applies for matrices.

To compute the dot product of two vectors (in other words, $\sum_i x_i y_i$), use the multiplication operator ‘*’ as follows:

```
>> x*y'
```

```
ans =
```

```
43
```

Note the single quote after y denotes the transpose of a vector or a matrix.

To compute an element-by-element multiplication of two vectors (or two arrays), use the following operator:

```
>> x .* y
```

```
ans =
```

```
4 15 24
```

That is, x .* y means $[1 \times 4, 3 \times 5, 4 \times 6] = [4 \quad 15 \quad 24]$.

Complex Numbers

LabVIEW MathScript supports complex numbers. The imaginary number is denoted with the symbol i or j, assuming that these symbols have not been used any other place in the program. It is critical to avoid such a symbol conflict for obtaining correct outcome. Enter the following and observe the outcomes:

```
>> z=3 + 4i % note the multiplication sign ‘*’ is not needed after 4
```

```
>> conj(z) % computes the conjugate of z
```

```
>> angle(z) % computes the phase of z
```

```
>> real(z) % computes the real part of z
```

```
>> imag(z) % computes the imaginary part of z
```

```
>> abs(z) % computes the magnitude of z
```

One can also define an imaginary number with any other user-specified variables. For example, try the following:

```
>> img=sqrt(-1)
```

```
>> z=3+4*img
```

```
>> exp(pi*img)
```

Array Indexing

In .m files, all arrays (vectors) are indexed starting from 1 − in other words, x(1) denotes the first element of the array x. Note that the arrays are indexed using parentheses (.) and not square brackets [.], as done in C/C++. To create an array featuring the integers 1 through 6 as elements, enter:

```
>> x=[1,2,3,4,5,6]
```

Alternatively, use the notation ‘:’

```
>> x=1:6
```

This notation creates a vector starting from 1 to 6, in steps of 1. If a vector from 1 to 6 in steps of 2 is desired, then type:

```
>> x=1:2:6
```

```
ans =
```

```
1 3 5
```

Also, examine the following code:

```
>> ii=2:4:17
```

```
>> jj=20:-2:0
```

```
>> ii=2:(1/10):4
```

One can easily extract numbers in a vector. To concatenate an array, the example below shows how to use the operator '[ ]':

```
>> x=[1:3 4 6 100:110]
```

To access a subset of this array, try the following:

```
>> x(3:7)
```

```
>> length(x) % gives the size of the array or vector
```

```
>> x(2:2:length(x))
```

Allocating Memory

One can allocate memory for one-dimensional arrays (vectors) using the command zeros. The following command allocates memory for a 100-dimensional array:

```
>> y=zeros(100,1);
```

```
>> y(30)
```

```
ans =
```

```
0
```

One can allocate memory for two-dimensional arrays (matrices) in a similar fashion. The command

```
>> y=zeros(4,5)
```

defines a 4 by 5 matrix. Similar to the command zeros, the command ones can be used to define a vector containing all ones,

```
>> y=ones(1,5)
```

```
ans=
```

```
1 1 1 1 1
```

Special Characters and Functions

Some common special characters used in .m files are listed below for later reference:

Symbol	Meaning
pi	$\pi(3.14.....)$
^	indicates power (for example, 3^2=9)
NaN	not-a-number, obtained when encountering undefined operations, such as $0/0$
Inf	Represents $+\infty$
;	indicates the end of a row in a matrix; also used to suppress printing on the screen (echo off)
%	comments − anything to the right of % is ignored by the .m file interpreter and is considered to be comments

‘	denotes transpose of a vector or a matrix; also used to define strings, for example, str1='DSP'
…	denotes continuation; three or more periods at the end of a line continue current function to next line

Some common special characters used in .m files

Some special functions are listed below for later reference:

<table>
<tr><th>Function</th><th>Meaning</th></tr>
<tr><td>sqrt</td><td>indicates square root, for example, sqrt(4)=2</td></tr>
<tr><td>abs</td><td>absolute value | . |, for example, abs(-3)=3</td></tr>
<tr><td>length</td><td>length(x) gives the dimension of the array x</td></tr>
<tr><td>sum</td><td>finds sum of the elements of a vector</td></tr>
<tr><td>find</td><td>finds indices of nonzero</td></tr>
</table>

Some common functions used in .m files

Here is an example of the function `length`,

```
>> x=1:10;

>> length(x)

ans =

10
```

The function find returns the indices of a vector that are non-zero. For example,

`I = find(x>4)` finds all the indices of x greater than 4. Thus, for the above example:

```
>> find(x> 4)
```

```
ans =
```

```
5 6 7 8 9 10
```

Control Flow

.m files have the following control flow constructs:

• if statements

• switch statements

• for loops

• while loops

• break statements

The constructs if, for, switch and while need to terminate with an end statement. Examples are provided below:

if

```
>> x=-3;
```

```
if x>0
```

```
str='positive'
```

```
elseif x<0
```

```
str='negative'

elseif x== 0

str='zero'

else

str='error'

end
```

See the value of 'str' after executing the above code.

while

```
>> x=-10;

while x<0

x=x+1;

end
```

See the value of x after executing the above code.

for loop

```
>> x=0;

for j=1:10

x=x+j;

end
```

The above code computes the sum of all the numbers from 1 to 10.

break

With the break statement, one can exit early from a for or a while loop:

```
>> x=-10;
while x<0
x=x+2;
if x = = -2
break;
end
end
```

LabVIEW MathScript supports the relational and logical operators listed below.

Relational Operators

Symbol	Meaning
<=	less than equal
<	less than
>=	greater than equal
>	greater than
==	equal

~=	not equal

Relational Operators

Logical Operators

Symbol	Meaning
&	AND
|	OR
~	NOT

Logical Operators

Programming in the LabVIEW MathScript Window

The MathScript feature allows one to include .m files, which can be created using any text editor. To activate the LabVIEW MathScript interactive window, select **Tools** → **MathScript Window** from the main menu. To open the LabVIEW MathScript text editor, click the Script tab of the LabVIEW MathScript Window (see [link]). After typing the .m file textual code, save it and click on the **Run script** button (green arrow) to run it.

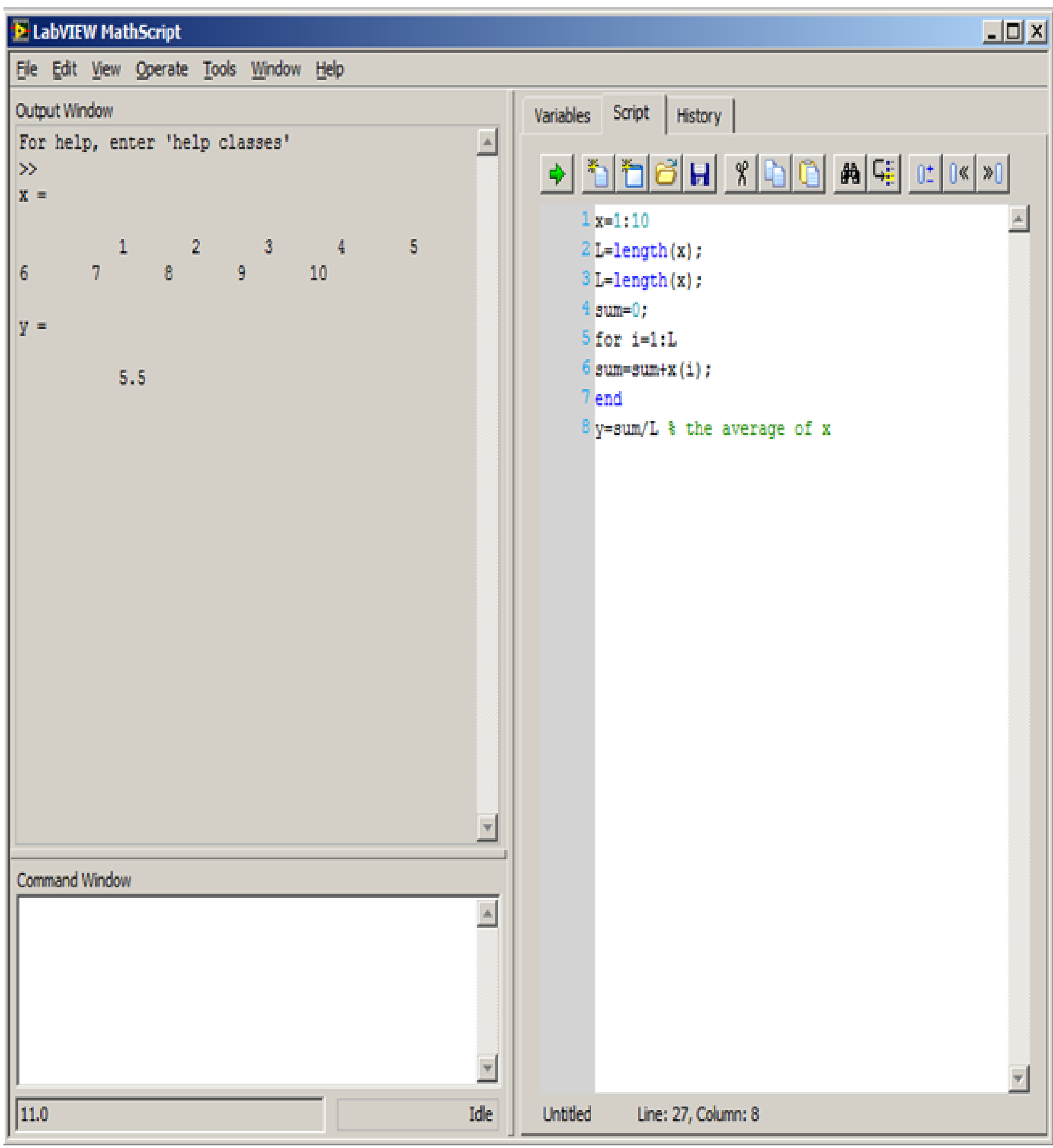
For instance, to write a program to compute the average (mean) of a vector x, the program should use as its input the vector x and return the average value. To write this program, follow the steps outlined below.

Type the following in the empty script:

```
x=1:10
```

```
L=length(x);
sum=0;
for j=1:L
sum=sum+x(j);
end
y=sum/L % the average of x
```

From the Editor pull-down menu, go to **File** → **Save Script As** and enter **average.m** for the file name. Then click on the **Run script** button to run the program. [link] shows the LabVIEW MathScript interactive window after running the program.



LabVIEW MathScript Interactive Window after Running the Program Average

Sound Generation

Assuming the computer used has a sound card, one can use the function sound to play back speech or audio files through its speakers. That is, sound(y,FS) sends the signal in a vector y (with sample frequency FS) out to the speaker. Stereo sounds are played on platforms that support them, with y being an N-by-2 matrix.

Try the following code and listen to a 400 Hz tone:

```
>> t=0:1/8000:1;
```

```
>> x=cos(2*pi*400*t);
```

```
>> sound(x,8000);
```

Now generate a noise signal by typing:

```
>> noise=randn(1,8000); % generate 8000 samples of noise
```

```
>> sound(noise,8000);
```

The function randn generates Gaussian noise with zero mean and unit variance.

Loading and Saving Data

One can load or store data using the commands load and save. To save the vector x of the above code in the file **data.mat**, type:

```
>> save data x
```

Note that LabVIEW MathScript data files have the extension .mat. To retrieve the data saved, type:

```
>> load data
```

The vector x gets loaded in memory. To see memory contents, use the command `whos`,

```
>> whos
```

```
Variable Dimension Type x   1x8000 double array
```

The command `whos` gives a list of all the variables currently in memory, along with their dimensions. In the above example, x contains 8000 samples.

To clear up memory after loading a file, type `clear all` when done. This is important because if one does not clear all the variables, one could experience conflicts with other programs using the same variables.

Reading Wave and Image Files

With LabVIEW MathScript, one can read data from different file types (such as .wav, .jpeg and .bmp) and load them in a vector.

To read an audio data file with .wav extension, use the following command:

```
>> [y Fs]=wavread('filename')
```

This command reads a wave file specified by the string filename and returns the sampled data in y with the sampling rate of Fs (in hertz).

To read an image file, use the following command:

```
>> [y]=imread('filename', 'filetype')
```

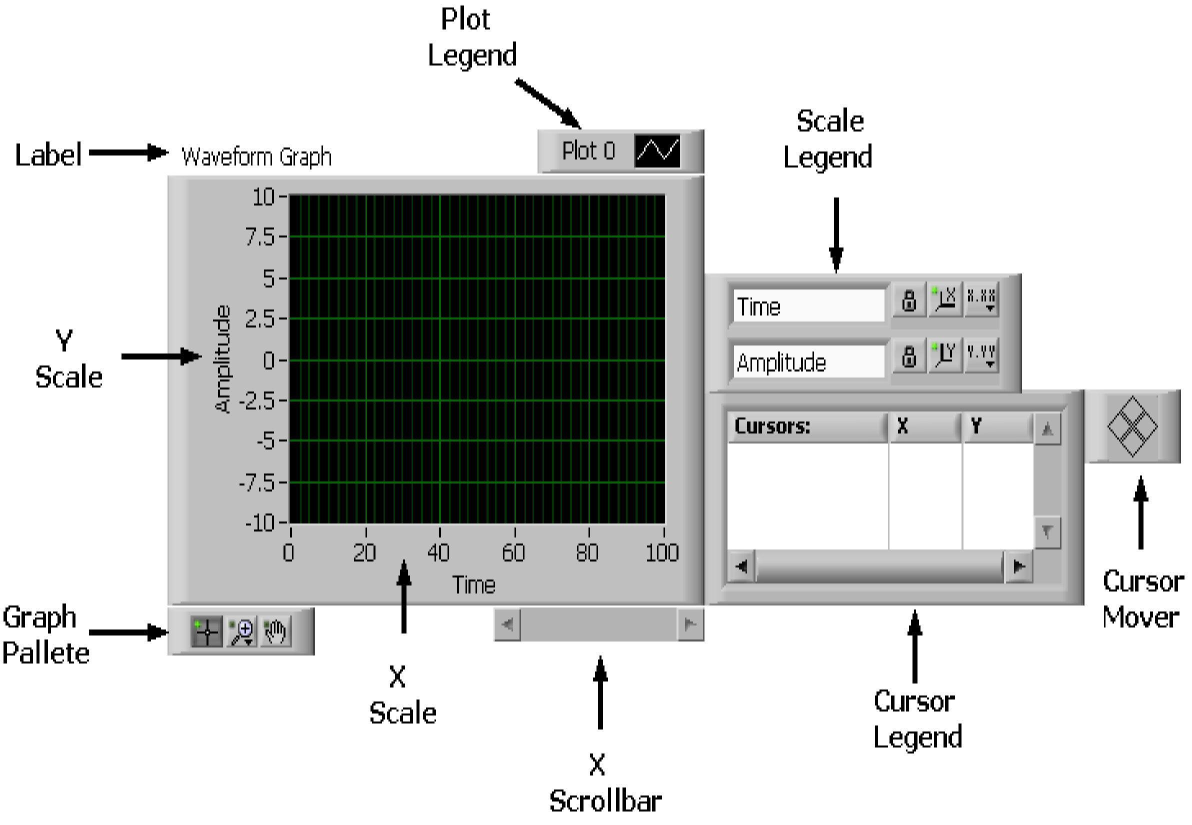
This command reads a grayscale or color image from the string filename, where filetype specifies the format of the file and returns the image data in the array y.

Signal Display

Several tools are available in LabVIEW to display data in a graphical format. Throughout the book, signals in both the time and frequency domains are displayed using the following two graph tools.

Waveform Graph—Displays data acquired at a constant rate.

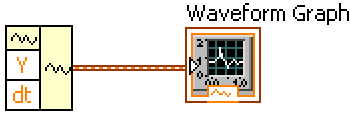
XY Graph—Displays data acquired at a non-constant rate, such as data acquired when a trigger occurs. A waveform graph can be created on a front panel by choosing **Controls**→ **Express** → **Waveform Graph**. [link] shows a waveform graph and the waveform graph elements which can be opened by right-clicking on the graph and selecting **Visible Items** from the shortcut menu.



Waveform Graph

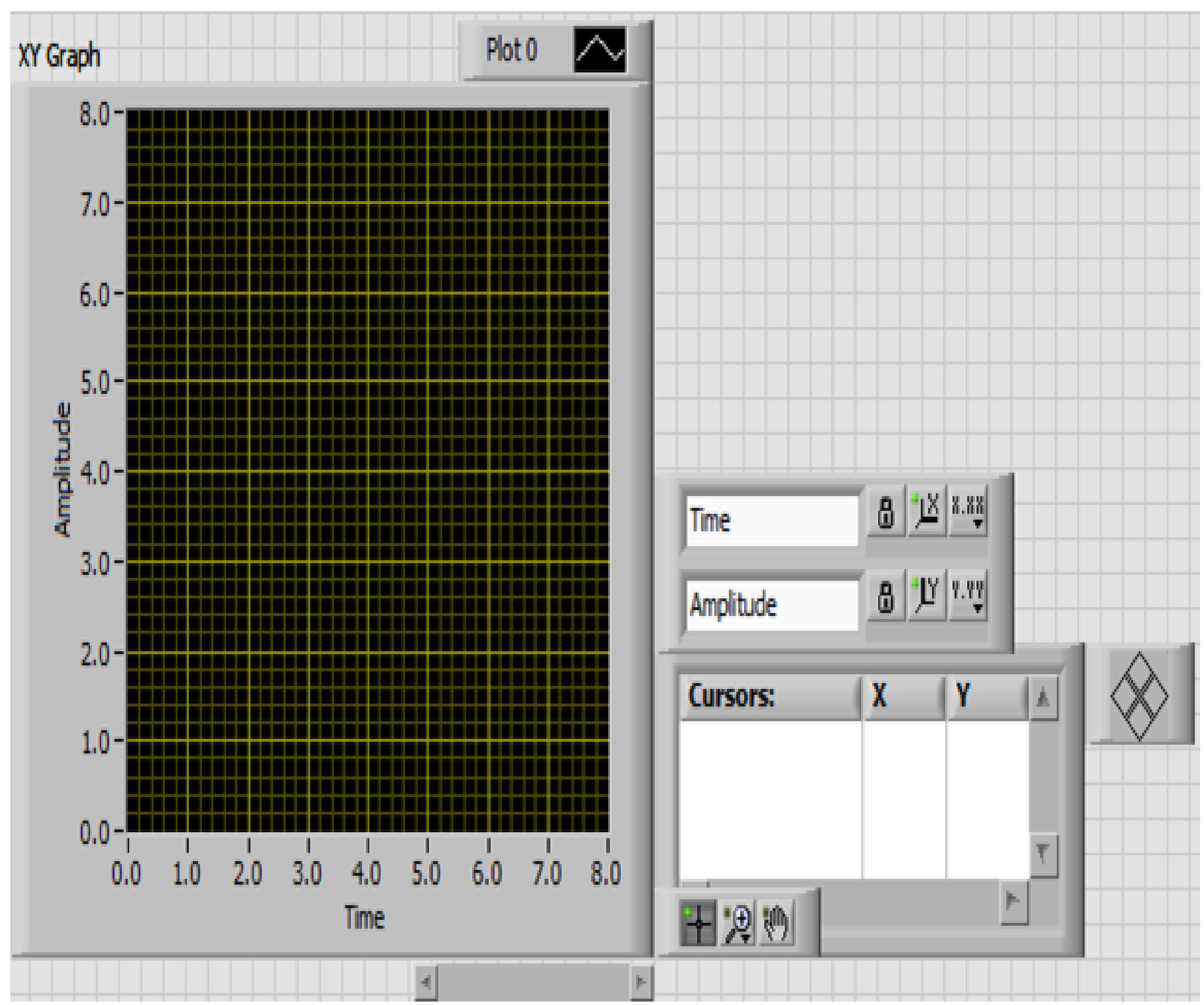
Often a waveform graph is tied with the function **Build Waveform(Function**→ **Programming** → **Waveform** → **Build**

Waveform) to calibrate the x scale (which is time scale for signals), as shown in [link].



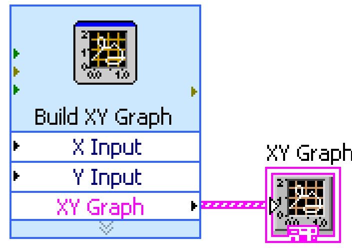
Build Waveform Function and Waveform Graph

Create an XY graph from a front panel by choosing **Controls**→ **Express** → **XY Graph**. [link] shows an XY graph and its different elements.



XY Graph

An XY graph displays a signal at a non-constant rate, and one can tie together its X and Y vectors to display the signal via the **Build XY Graph** function. This function automatically appears on the block diagram when placing an **XY graph** on the front panel, as shown in [link]. Note that one can use the function Bundle **(Functions → Programming → Cluster & Variant → Bundle)** instead of **Build XY Graph**.



Build XY Graph Function

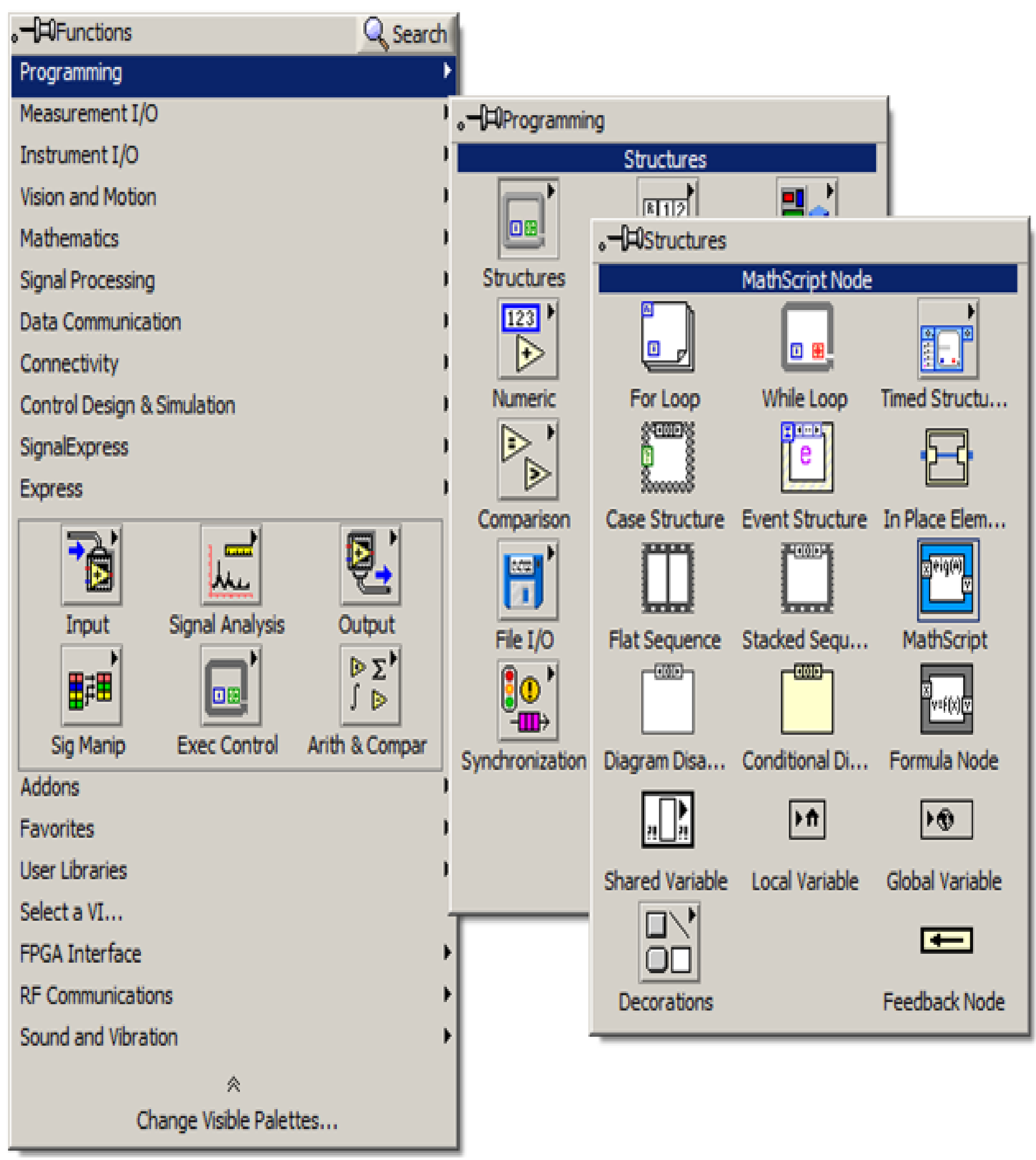
Hybrid Programming

As stated earlier, the LabVIEW MathScript feature can be used to perform hybrid programming, in other words, a combination of textual .m files and graphical objects. Normally, it is easier to carry out math operations via .m files while maintaining user interfacing, interactivity and analysis in the more intuitive graphical environment of LabVIEW. Textual .m file codes can be typed in or copied and pasted into LabVIEW MathScript nodes.

Sum and Average VI Example Using Hybrid Programming

Sum and Average VI Example Using Hybrid Programming

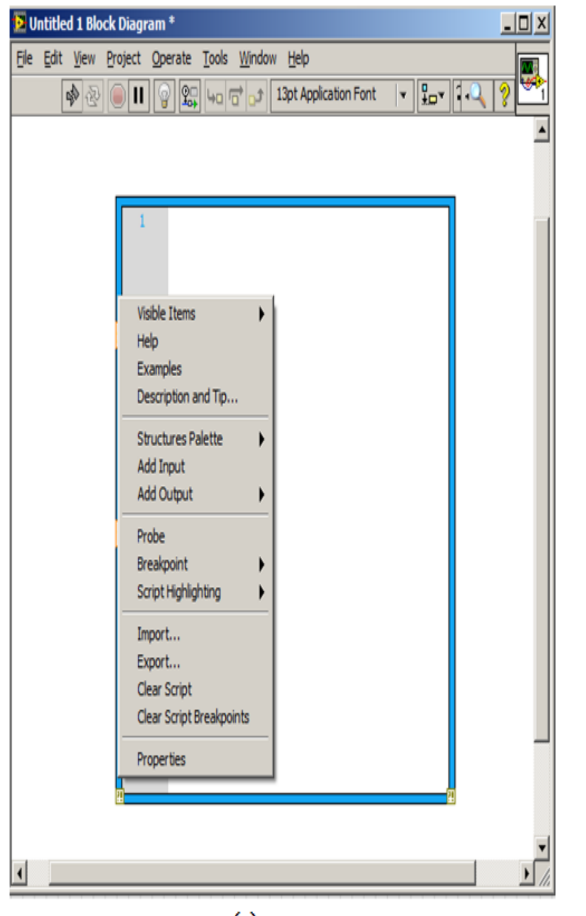
Choose **Functions** →**Programming** →**Structures** → **MathScript** to create a LabVIEW MathScript node (see [link]). Change the size of the window by dragging the mouse.

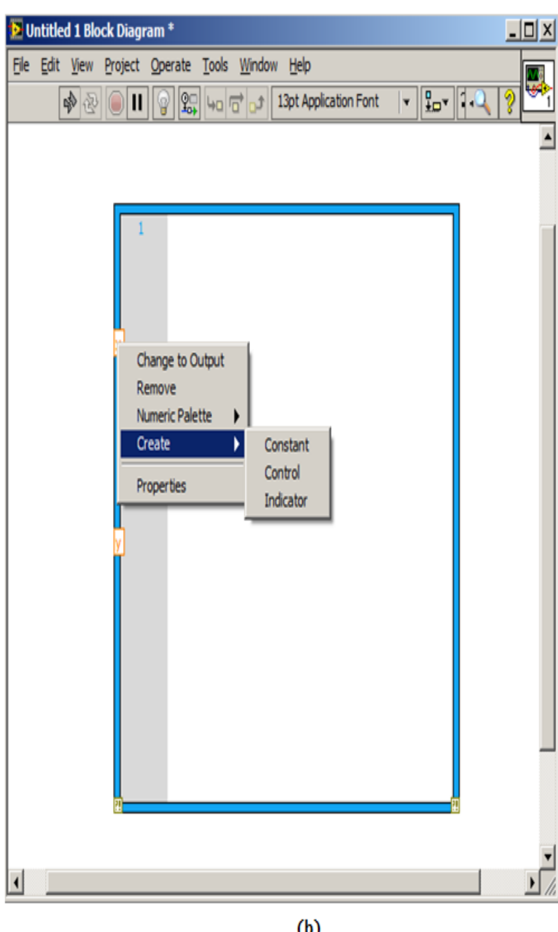


LabVIEW MathScript Node Creation

Now build the same program average using a LabVIEW MathScript node. The inputs to this program consist of x and y. To add these inputs, right-

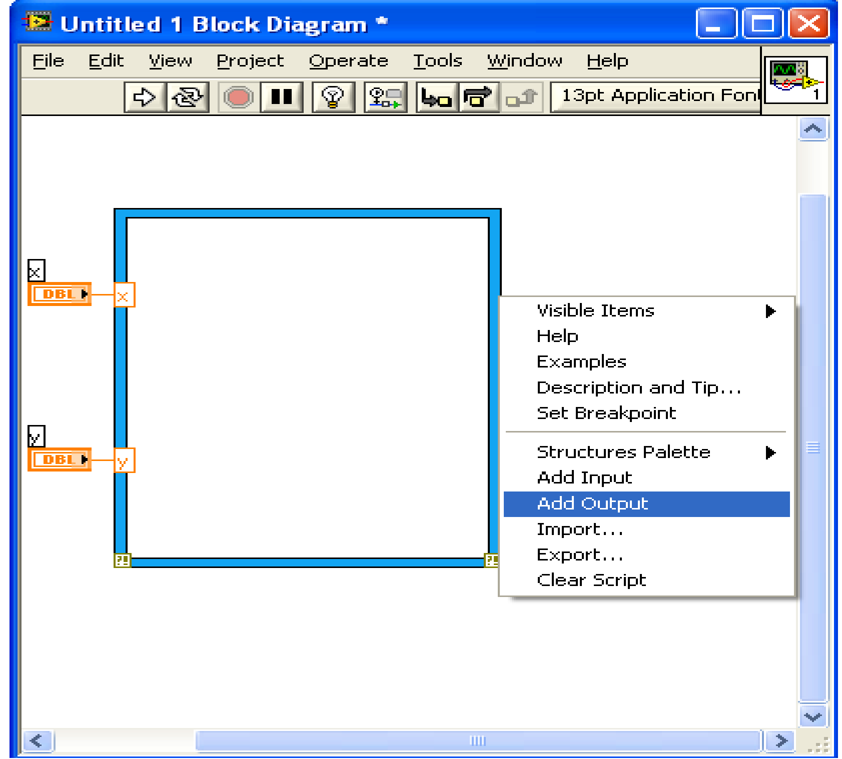
click on the border of the LabVIEW MathScript node and click on the Add Input option (see [link]).



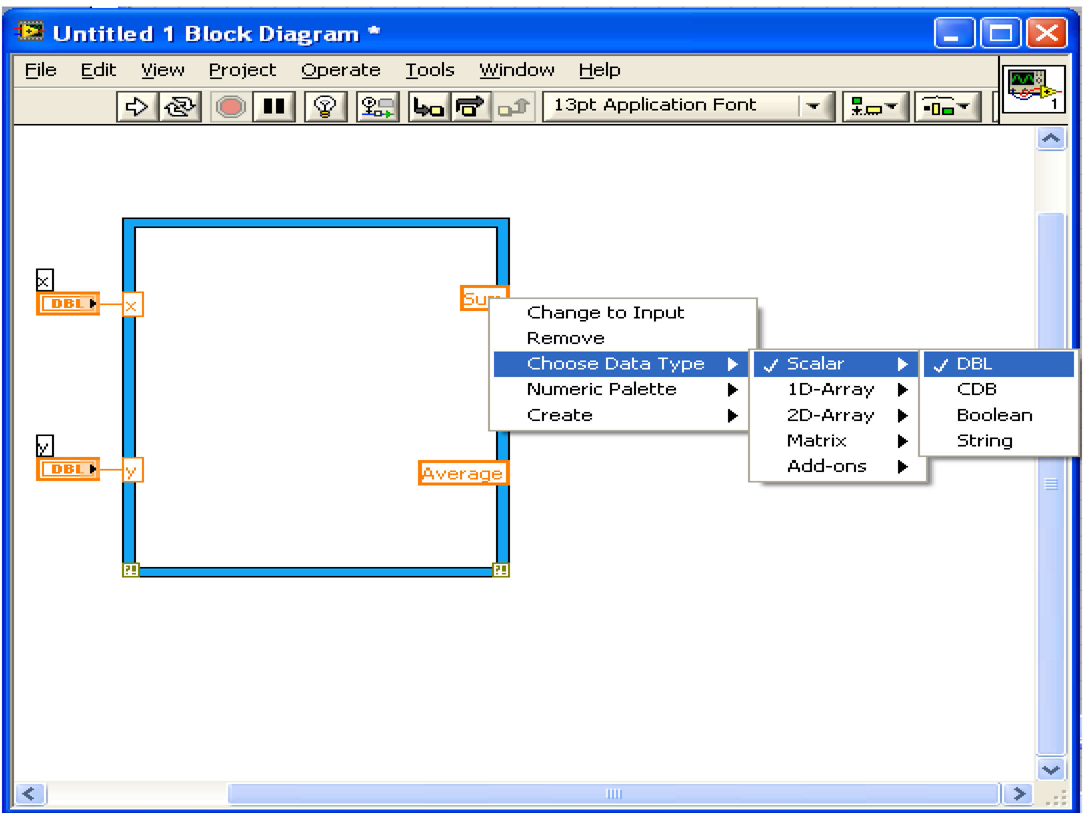


(a) Adding Inputs, (b) Creating Controls

After adding these inputs, create controls to change the inputs interactively via the front panel. By right-clicking on the border, add outputs in a similar manner. An important issue to consider is the selection of output data type. The outputs of the Sum and Average VI are scalar quantities. Choose data types by right-clicking on an output and selecting the Choose Data Type option (see [link]).



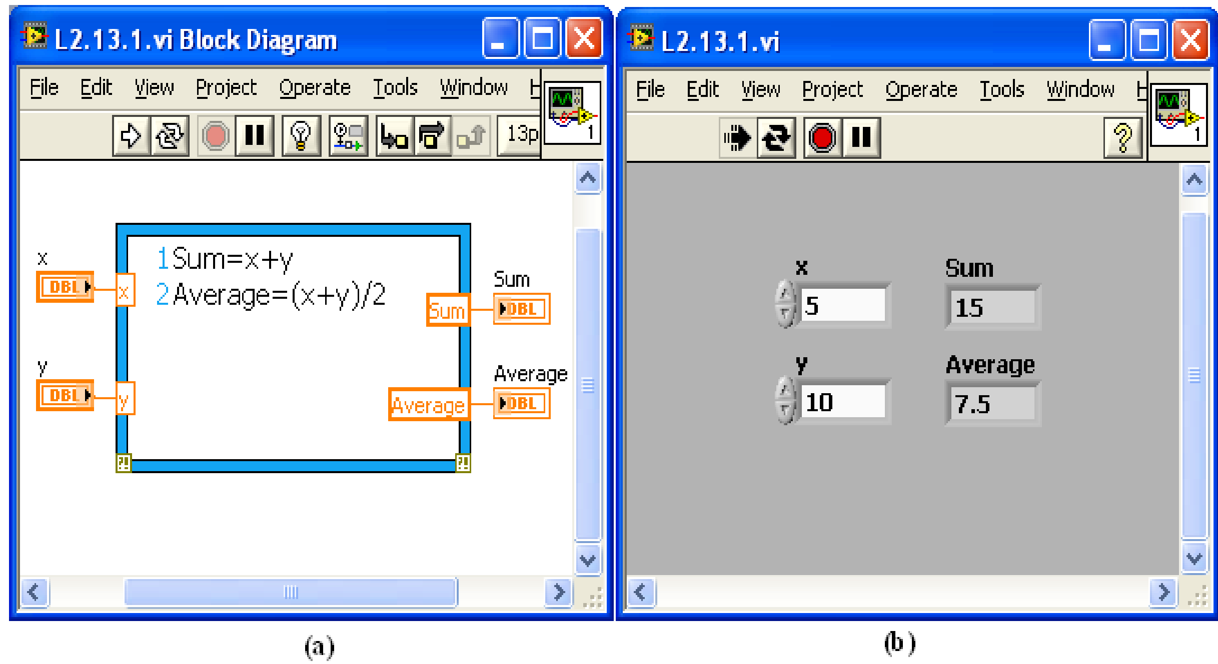
(a)



(b)

(a) Adding Outputs, (b) Choosing Data Types

Finally, add numeric indicators in a similar fashion as indicated earlier. [link] shows the completed block diagram and front panel.

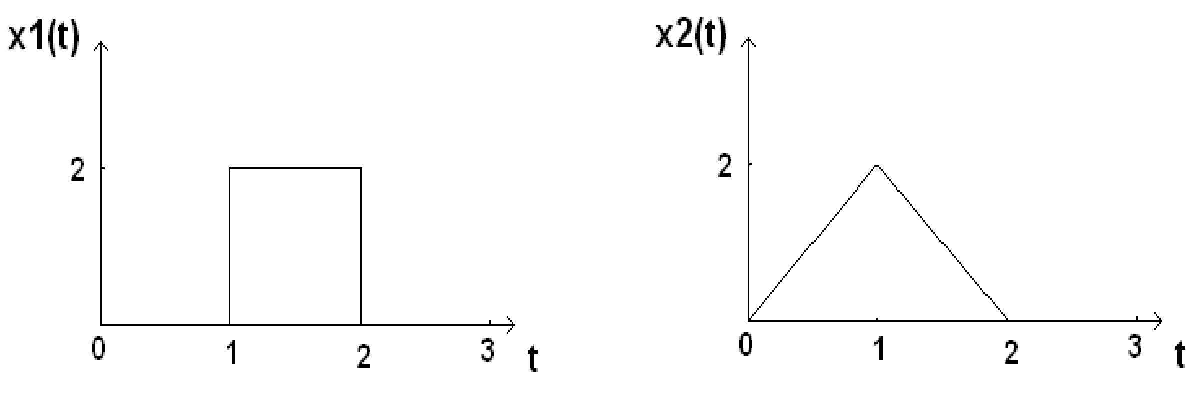


(a) Completed Block Diagram, (b) Completed Front Panel

Building a Signal Generation System Using Hybrid Programming

In this section, let us see how to generate and display aperiodic continuous-time signals or pulses in the time domain. One can represent such signals with a function of time. For simulation purposes, a representation of time t is needed. Note that the time scale is continuous while computer programs

operate in a discrete fashion. This simulation can be achieved by considering a very small time interval. For example, if a 1-second duration signal in millisecond increments (time interval of 0.001 second) is considered, then one sample every 1 millisecond and a total of 1000 samples are generated for the entire signal. This continuous-time signal approximation is discussed further in later chapters. It is important to note that there is a finite number of samples for a continuous-time signal, and, to differentiate this signal from a discrete-time signal, one must assign a much higher number of samples per second (very small time interval).



Continuous-Time Signals

[link] shows two continuous-time signals $x1(t)$ and $x2(t)$with a duration of 3 seconds. By setting the time interval dt to 0.001 second, there is a total of 3000 samples at $t=$ 0,0.001,0.002,0.003,.......,2.999 seconds.

The signal $x1(t)$ can be represented mathematically as follows:
Equation:

$$x1(t)=\begin{matrix} 0 & 0\leq t<1 \\ 1 & 1\leq t<2 \\ 0 & 2\leq t<3 \end{matrix}$$

To simulate this signal, use the LabVIEW MathScript functions `ones` and `zeros`. The signal value is zero during the first second, which means the first 1000 samples are zero. This portion of the signal is simulated with the function `zeros(1,1000)`. In the next second (next 1000 samples), the signal value is 2, and this portion is simulated by the function `2*ones(1,1000)`. Finally, the third portion of the signal is simulated by the function `zeros(1,1000)`. In other words, the entire duration of the signal is simulated by the following .m file function:

```
x1=[ zeros(1,1/dt) 2*ones(1,1/dt) zeros(1,1/dt)]
```

The signal $x2(t)$ can be represented mathematically as follows:
Equation:

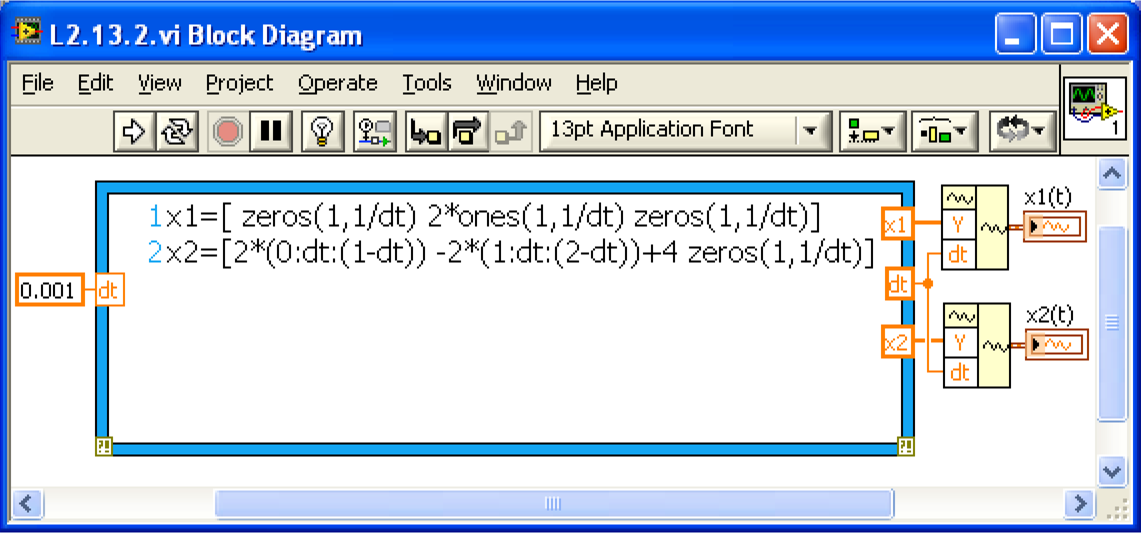
$$x2(t) = \begin{array}{cc} 2\text{t} & 0 \le t < 1 \\ -2\text{t} + 4 & 1 \le t < 2 \\ 0 & 2 \le t < 3 \end{array}$$

Use a linearly increasing or decreasing vector to represent the linear portions. The time vectors for the three portions or segments of the signal are `0:dt:1-dt`, `1:dt:2-dt` and `2:dt:3-dt`. The first segment is a linear function corresponding to a time vector with a slope of 2; the second segment is a linear function corresponding to a time vector with a slope of -2 and an offset of 4; and the third segment is simply a constant vector of zeros. In other words, simulate the entire duration of the signal for any value of dt by the following .m file function:

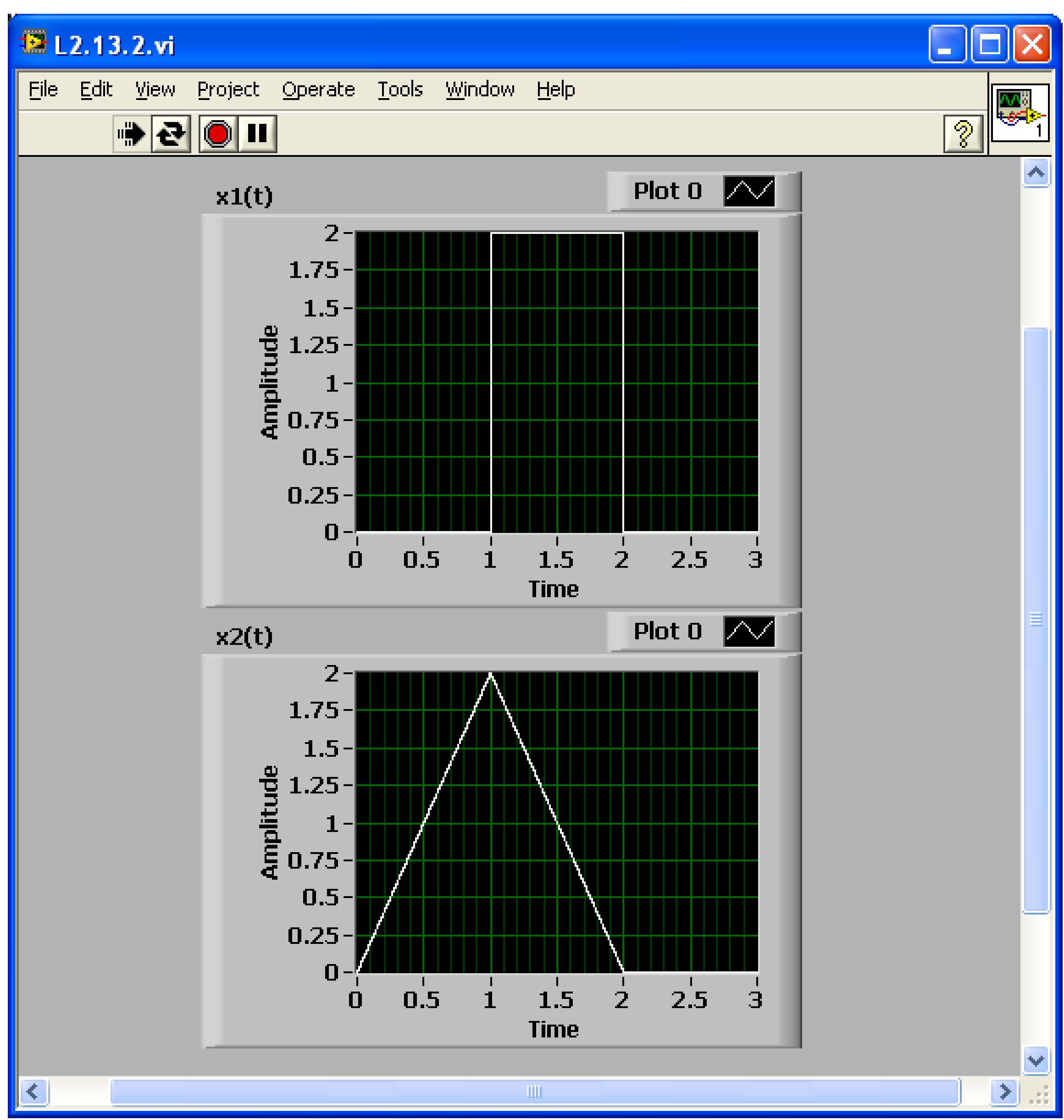
```
x2=[2*(0:dt:(1-dt)) -2*(1:dt:(2-dt))+4
zeros(1,1/dt)].
```

[link] and [link] show the block diagram and front panel of the above signal generation system, respectively. Display the signals using a **Waveform Graph(Controls→ Express → Waveform Graph)** and a **Build Waveform** function **(Function→ Programming → Waveform → Build Waveform)**. Note that the default data type in MathScript is double precision scalar. So whenever an output possesses any other data type, one

needs to right-click on the output and select the **Choose Data Type** option. In this example, x1 and x2 are double precision one-dimensional arrays that are specified accordingly.



Block Diagram of a Signal Generation System



Front Panel of a Signal Generation System

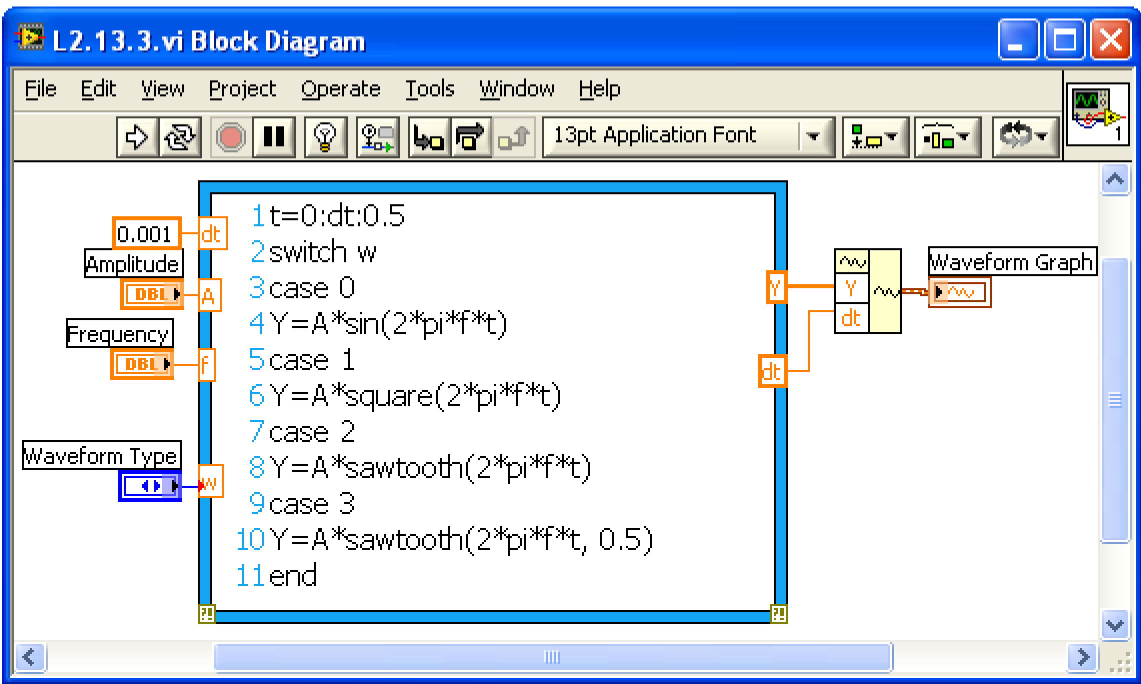
Building a Periodic Signal Generation System Using Hybrid Programming

In this section, build a simple periodic signal generation system in hybrid mode to set the stage for the chapters that follow. This system involves generating a periodic signal in textual mode and displaying it in graphical mode. Modify the shape of the signal (sine, square, triangle or sawtooth) as well as its frequency and amplitude by using appropriate front panel controls. The block diagram and front panel of this system using a LabVIEW MathScript node are shown in [link] and [link], respectively. The front panel includes the following three controls:

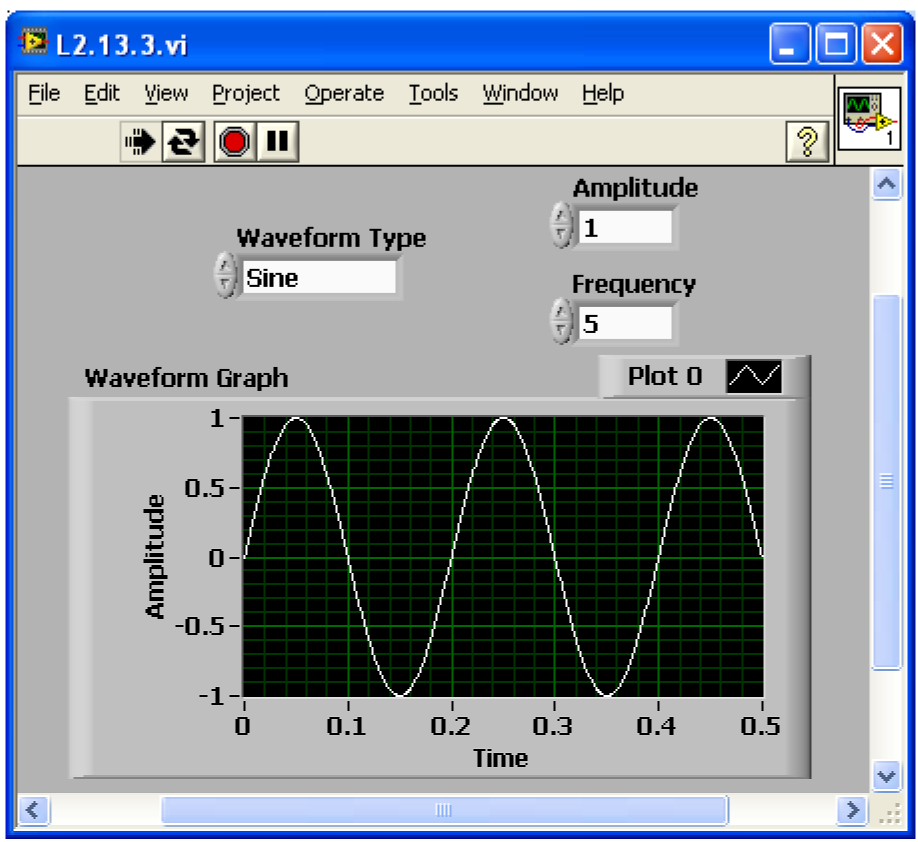
Waveform type – Select the shape of the input waveform as either sine, square, triangular or sawtooth waves.

Amplitude – Control the amplitude of the input waveform.

Frequency – Control the frequency of the input waveform.



Periodic Signal Generation System Block Diagram

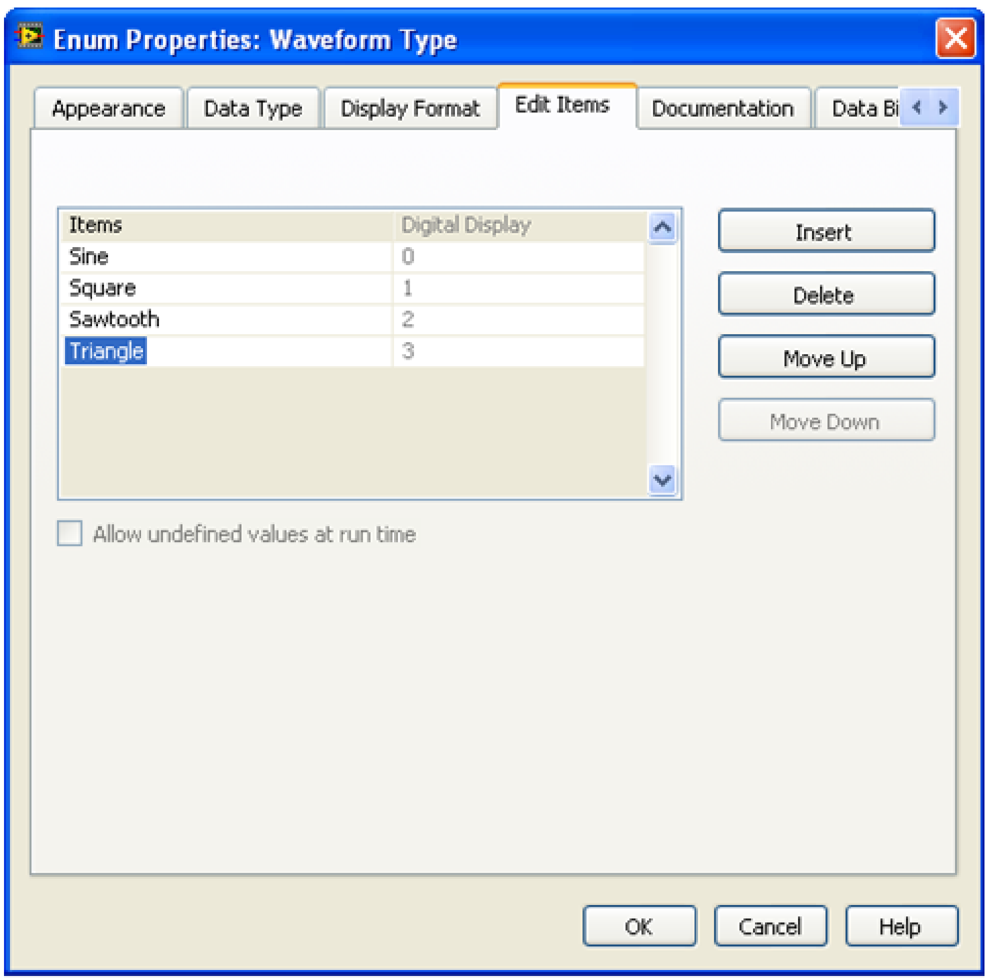


Periodic Signal Generation System Front Panel

To build the block diagram, first write a .m file code to generate four types of waveforms using the .m file functions `sin`, `square` and `sawtooth`. To change the amplitude and frequency of the waveforms, use two controls named Amplitude (A) and Frequency (f). Waveform Type (w) is another input controlled by the **Enum Control** for selecting the waveform type. With this control, one can select from multiple inputs. Create an Enum Control from the front panel by invoking **Controls** → **Modern** → **Ring & Enum** → **Enum**. Right-click on the **Enum Control** to select **properties** and the **edit item** tab to choose different items as shown in [link]. After inserting each item, the digital display shows the corresponding number value for that item, which is the output of the **Enum Control**.

Finally, display the waveforms with a **Waveform Graph(Controls→ Express** → **Waveform Graph)** and a **Build Waveform** function

(Function→ Programming → Waveform → Build Waveform).



Enum Control Properties

Lab Exercises

Exercise:

Problem:

Write a .m file code to add all the numbers corresponding to the even indices of an array. For instance, if the array x is specified as x = [1, 3, 5, 10], then 13 (= 3+10) should be returned. Use the program to find the sum of all even integers from 1 to 1000. Run your code using the LabVIEW MathScript interactive window. Also, redo the code where x is the input vector and y is the sum of all the numbers corresponding to the even indices of x.

Solution:

Insert Solution Text Here

Exercise:

Problem: 2. Explain what the following .m file does:

```
L=length(x);

for j=1:L

if x(j) < 0

x(j)=-x(j);

end

end
```

Rewrite this program without using a `for` loop.

Solution:

Insert Solution Text Here

Exercise:

Problem:

3. Write a .m file code that implements the following hard-limiting function:

Equation:

$$x(t) = \begin{cases} 0.2 & t \geq 0.2 \\ -0.2 & t < 0.2 \end{cases}$$

For t, use 1000 random numbers generated via the function rand.

Solution:

Insert Solution Text Here

Exercise:

Problem:

4. Build a hybrid VI to generate two sinusoid signals with the frequencies f1 Hz and f2 Hz and the amplitudes A1 and A2, based on a sampling frequency of 8000 Hz with the number of samples being 256. Set the frequency ranges from 100 to 400 Hz and set the amplitude ranges from 20 to 200. Generate a third signal with the frequency f3 = (mod (lcm (f1, f2), 400) + 100) Hz, where mod and lcm denote the modulus and least common multiple operation, respectively, and the amplitude A3 is the sum of the amplitudes A1 and A2. Use the same sampling frequency and number of samples as specified for the first two signals. Display all the signals using the legend on the same waveform graph and label them accordingly.

Solution:

Insert Solution Text Here

Convolution and Its Numerical Approximation

The output $y(t)$ of a continuous-time linear time-invariant (LTI) system is related to its input $x(t)$ and the system impulse response $h(t)$ through the convolution integral expressed as (for details on the theory of convolution and LTI systems, refer to signals and systems textbooks, for example, references [link] - [link]):

Equation:

$$y(t)=\int_{-\infty}^{\infty}h(t-\tau)x(\tau)d\tau$$

For a computer program to perform the above continuous-time convolution integral, a numerical approximation of the integral is needed noting that computer programs operate in a discrete – not continuous – fashion. One way to approximate the continuous functions in the Equation (1) integral is to use piecewise constant functions. Define $\delta_{\Delta}(t)$ to be a rectangular pulse of width Δ and height 1, centered at $t=0$:

Equation:

$$\delta_{\Delta}(t)=\begin{cases}1 & -\Delta/2\leq t\leq\Delta/2\\ 0 & \text{otherwise}\end{cases}$$

Approximate a continuous function $x(t)$ with a piecewise constant function $x_{\Delta}(t)$ as a sequence of pulses spaced every Δ seconds in time with heights $x(k\Delta)$:

Equation:

$$x_{\Delta}(t)=\sum_{k=-\infty}^{\infty}x(k\Delta)\delta_{\Delta}(t-k\Delta)$$

It can be shown in the limit as $\Delta \to 0, x_\Delta(t) \to x(t)$. As an example, [link] shows the approximation of a decaying exponential $x(t) = \exp(-\frac{t}{2})$ starting from 0 using $\Delta = 1$. Similarly, $h(t)$ can be approximated by

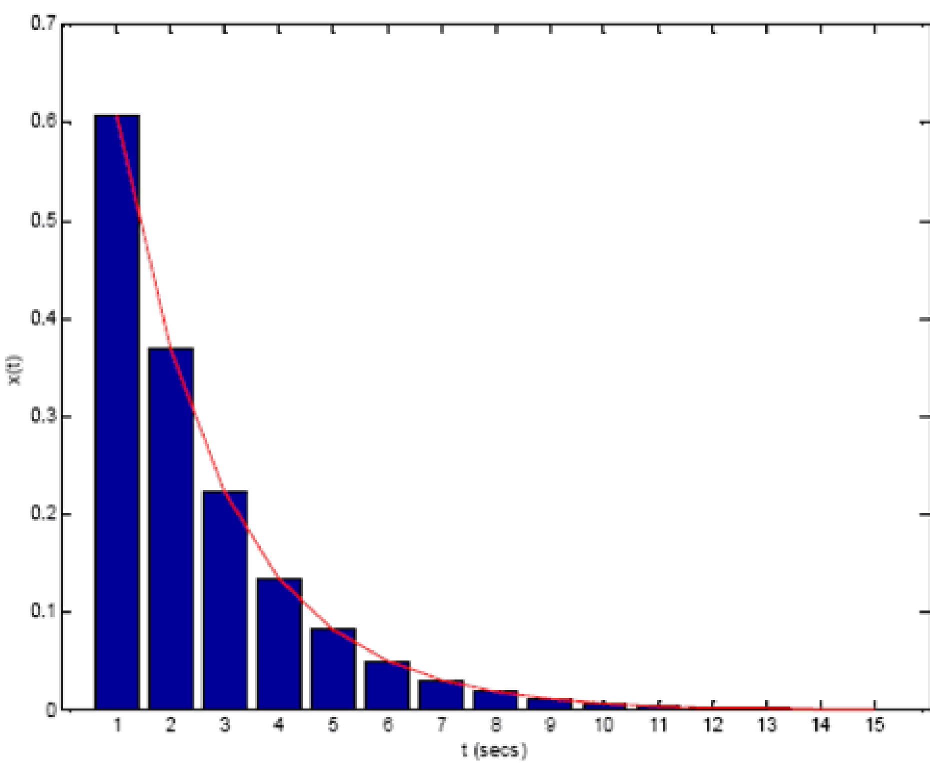
Equation:

$$h_\Delta(t) = \sum_{k=-\infty}^{\infty} h(k\Delta)\delta_\Delta(t - k\Delta)$$

One can thus approximate the convolution integral by convolving the two piecewise constant signals as follows:

Equation:

$$y_\Delta(t) = \int_{-\infty}^{\infty} h_\Delta(t-\tau)x_\Delta(\tau)d\tau$$



Approximation of a Decaying Exponential with Rectangular Strips of Width 1

Notice that $y_{\Delta}(t)$ is not necessarily a piecewise constant. For computer representation purposes, discrete output values are needed, which can be obtained by further approximating the convolution integral as indicated below:

Equation:

$$y_{\Delta}(n\Delta) = \Delta \sum_{k=-\infty}^{\infty} x(k\Delta)h((n-k)\Delta)$$

If one represents the signals $h_{\Delta}(t)$ and $x_{\Delta}(t)$ in a .m file by vectors containing the values of the signals at $t = n\Delta$, then Equation (5) can be used to compute an approximation to the convolution of $x(t)$ and $h(t)$.

Compute the discrete convolution sum $\sum_{k=-\infty}^{\infty} x(k\Delta)h((n-k)\Delta)$with the built-in LabVIEW MathScript command `conv`. Then, multiply this sum by Δ to get an estimate of $y(t)$ at $t = n\Delta$ Note that as Δ is made smaller, one gets a closer approximation to $y(t)$.

Convolution Properties

Convolution satisfies the following three properties (see [link]):

- Commutative property

Equation:

$$x(t) * h(t) = h(t) * x(t)$$

- Associative property

Equation:

$$x(t) * h_1(t) * h_2(t) = x(t) * \{h_1(t) * h_2(t)\}$$

- Distributive property

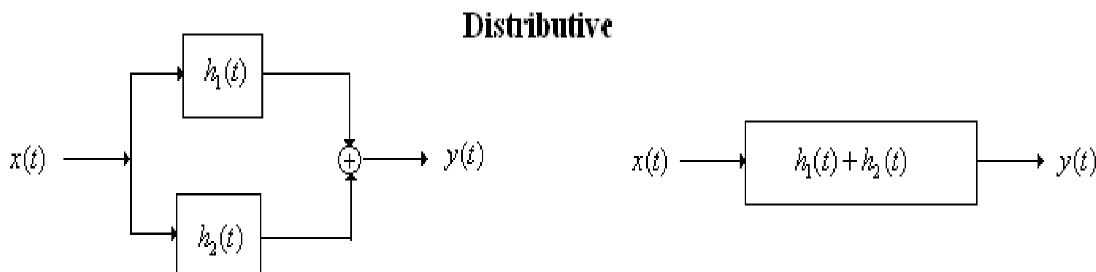
Equation:

$$x(t) * \{h_1(t) + h_2(t)\} = x(t) * h_1(t) + x(t) * h_2(t)$$

Commutative

Associative

Distributive



Convolution Properties

Lab 3: Convolution and Its Applications

This lab involves experimenting with the convolution of two continuous-time signals. The main mathematical part is written as a .m file, which is then used as a LabVIEW MathScript node within the LabVIEW programming environment to gain user interactivity. Due to the discrete-time nature of programming, an approximation of the convolution integral is needed. As an application of the convolution concept, echoes are removed from speech recordings using this concept.

Numerical Approximation of Convolution

In this section, let us apply the LabVIEW MathScript function `conv` to compute the convolution of two signals. One can choose various values of the time interval Δ to compute numerical approximations to the convolution integral.

Convolution Example 1

In this example, use the function conv to compute the convolution of the signals $x(t) = \exp(-\mathrm{at})u(t)$ and $h(t) = \exp(-\mathrm{bt})u(t)$with $u(t)$representing a step function starting at 0 for $0 \leq t \leq 8$. Consider the following values of the approximation pulse width or delta: $\Delta = 0.5, 0.1, 0.05, 0.01, 0.005, 0.001$. Mathematically, the convolution of $h(t)$and $x(t)$is given by

Equation:

$$y(t) = \frac{1}{a-b}(e^{-\mathrm{bt}} - e^{-\mathrm{at}})u(t)$$

Compare the approximation $\hat{y}(n\Delta)$obtained via the function `conv` with the theoretical value $y(t)$given by Equation (1). To better see the difference between the approximated $\hat{y}(n\Delta)$and the true $\hat{y}(n\Delta)$values, display $\hat{y}(t)$and $y(t)$ in the same graph.

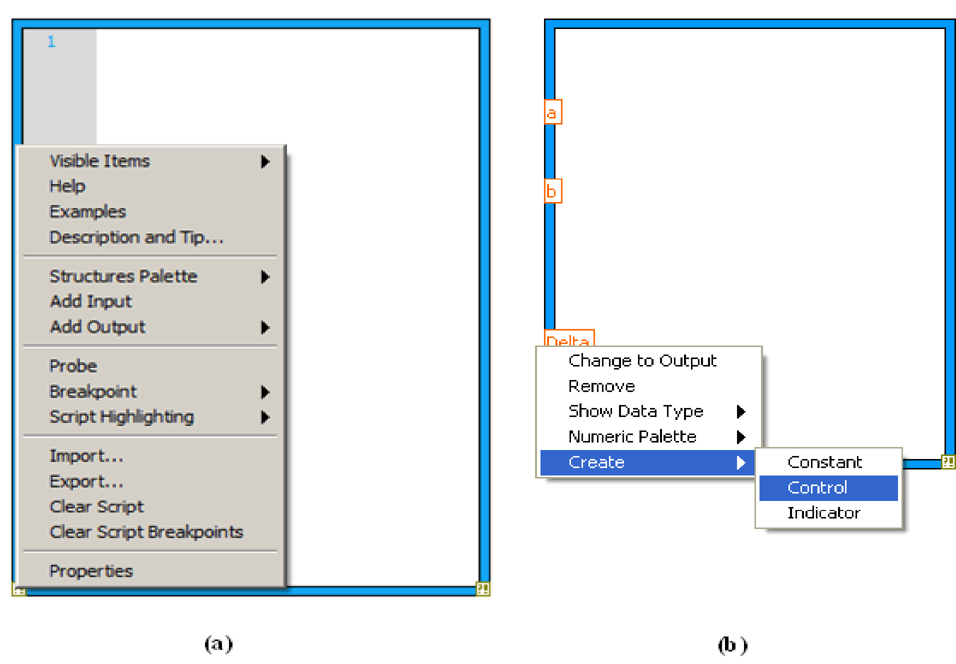
Compute the mean squared error (MSE) between the true and approximated values using the following equation:

Equation:

$$\mathrm{MSE} = \frac{1}{N}\sum_{n=1}^{N}(y(n\Delta) - \hat{y}(n\Delta))^2$$

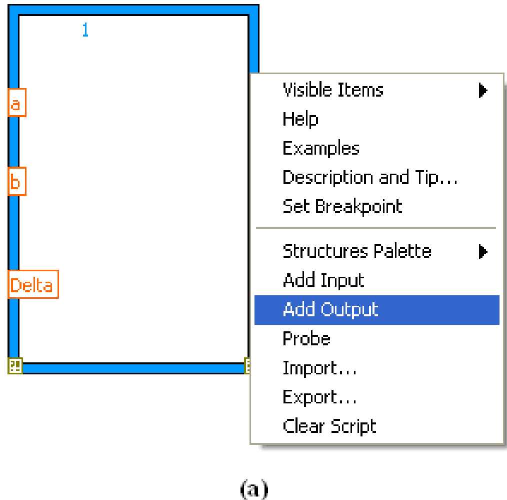
where $N = \lfloor \frac{T}{\Delta} \rfloor$, T is an adjustable time duration expressed in seconds and the symbol $\lfloor . \rfloor$ denotes the nearest integer. To begin with, set $T = 8$.

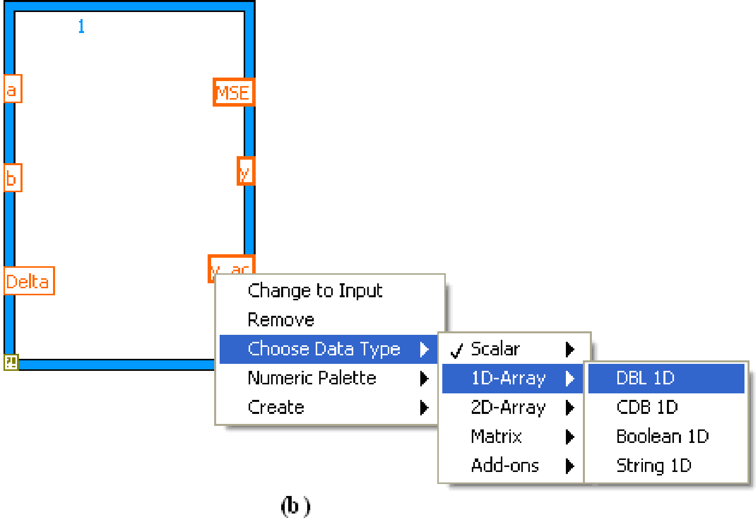
As you can see here, the main program is written as a .m file and placed inside LabVIEW as a LabVIEW MathScript node by invoking **Functions** → **Programming** →**Structures** → **MathScript**. The .m file can be typed in or copied and pasted into the LabVIEW MathScript node. The inputs to this program consist of an approximation pulse width Δ, input exponent powers $a$and b and a desired time duration T. To add these inputs, right-click on the border of the LabVIEW MathScript node and click on the **Add Input** option as shown in [link].



(a) Adding Inputs, (b) Creating Controls

After adding these inputs, create controls to allow one to alter the inputs interactively via the front panel. By right-clicking on the border, add the outputs in a similar manner. An important consideration is the selection of the output data type. Set the outputs to consist of MSE, actual or true convolution output y_ac and approximated convolution output y. The first output is a scalar quantity while the other two are one-dimensional vectors. The output data types should be specified by right-clicking on the outputs and selecting the **Choose Data Type** option (see [link]).





(a) Adding Outputs, (b) Choosing Data Types

Next write the following .m file textual code inside the LabVIEW MathScript node:

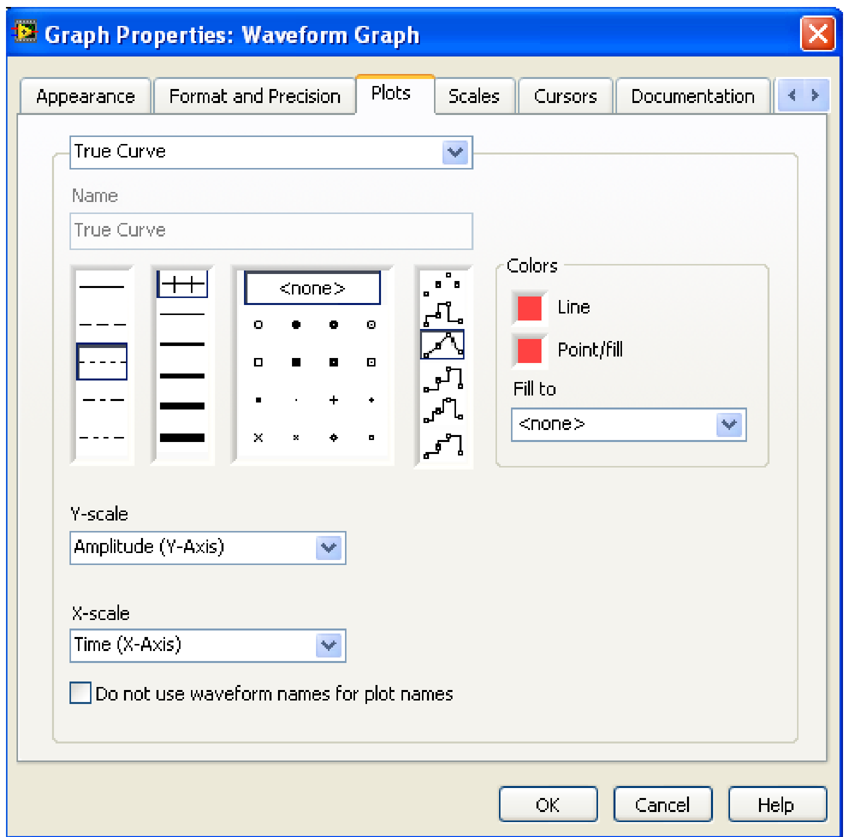
```
t=0:Delta:8;

Lt=length(t);

x1=exp(-a*t);

x2=exp(-b*t);
```

```
y=Delta*conv(x1,x2);

y_ac=1/(a-b)*(exp(-b*t)-exp(-a*t));

MSE=sum((y(1:Lt)-y_ac).^2)/Lt
```

With this code, a time vector t is generated by taking a time interval of Delta for 8 seconds. Convolve the two input signals, x1 and x2, using the function conv. Compute the actual output y_ac using Equation (1). Measure the length of the time vector and input vectors by using the command length(t). The convolution output vector y has a different size (if two input vectors m and n are convolved, the output vector size is m+n-1). Thus, to keep the size the same, use a portion of the output corresponding to y(1:Lt) during the error calculation.

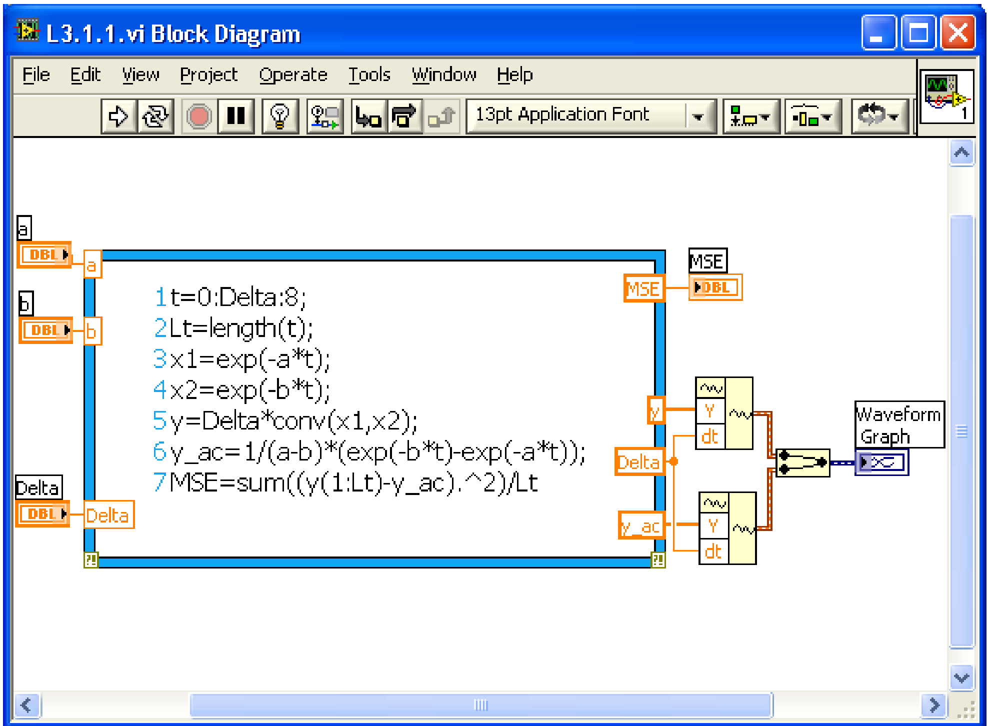
Use a waveform graph to show the waveforms. With the function Build Waveform **(Functions → Programming → Waveforms → Build Waveforms)**, one can show the waveforms across time. Connect the time interval Delta to the input dt of this function to display the waveforms along the time axis (in seconds).

Merge together and display the true and approximated outputs in the same graph using the function Merge Signal **(Functions → Express → Sig Manip → Merge Signals)**. Configure the properties of the waveform graph as shown in [link].



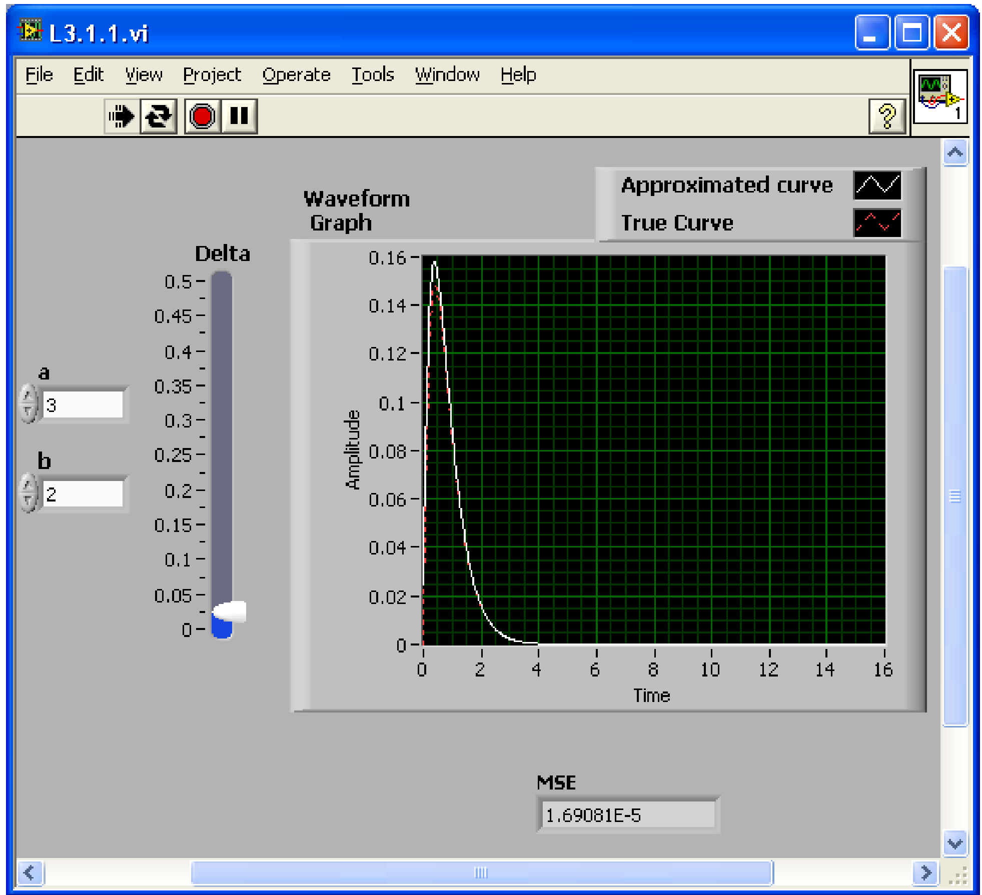
Waveform Graph Properties Dialog Box

[link] illustrates the completed block diagram of the numerical convolution.



Block Diagram of the Convolution Example

[link] shows the corresponding front panel, which can be used to change parameters. Adjust the input exponent powers and approximation pulse-width **Delta** to see the effect on the **MSE**.



Front Panel of the Convolution Example

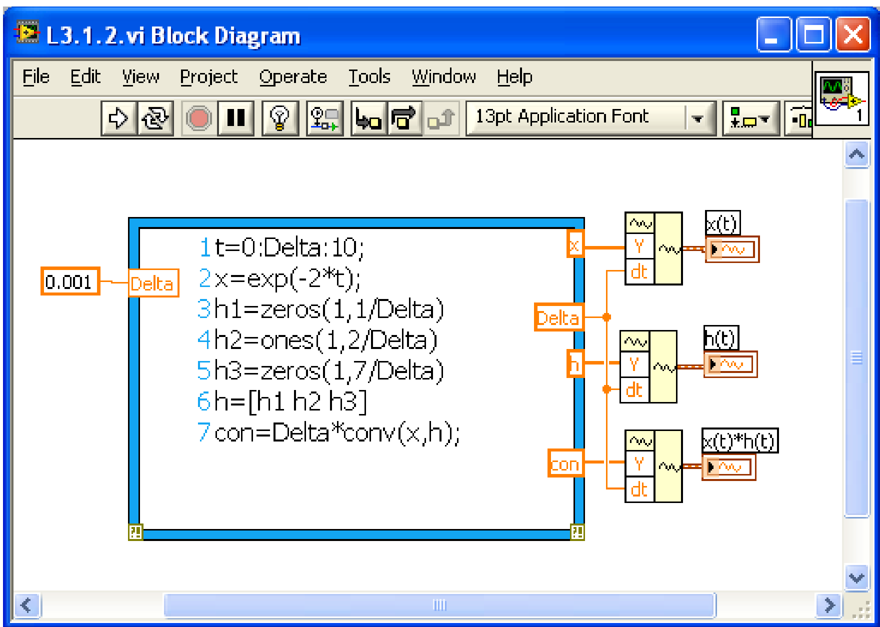
Convolution Example 2

Next, consider the convolution of the two signals $x(t) = \exp(-2\text{t})u(t)$and $h(t) = \text{rect}(\frac{t-2}{2})$ for , where $u(t)$denotes a step function at time 0 and `rect` a rectangular function defined as

Equation:

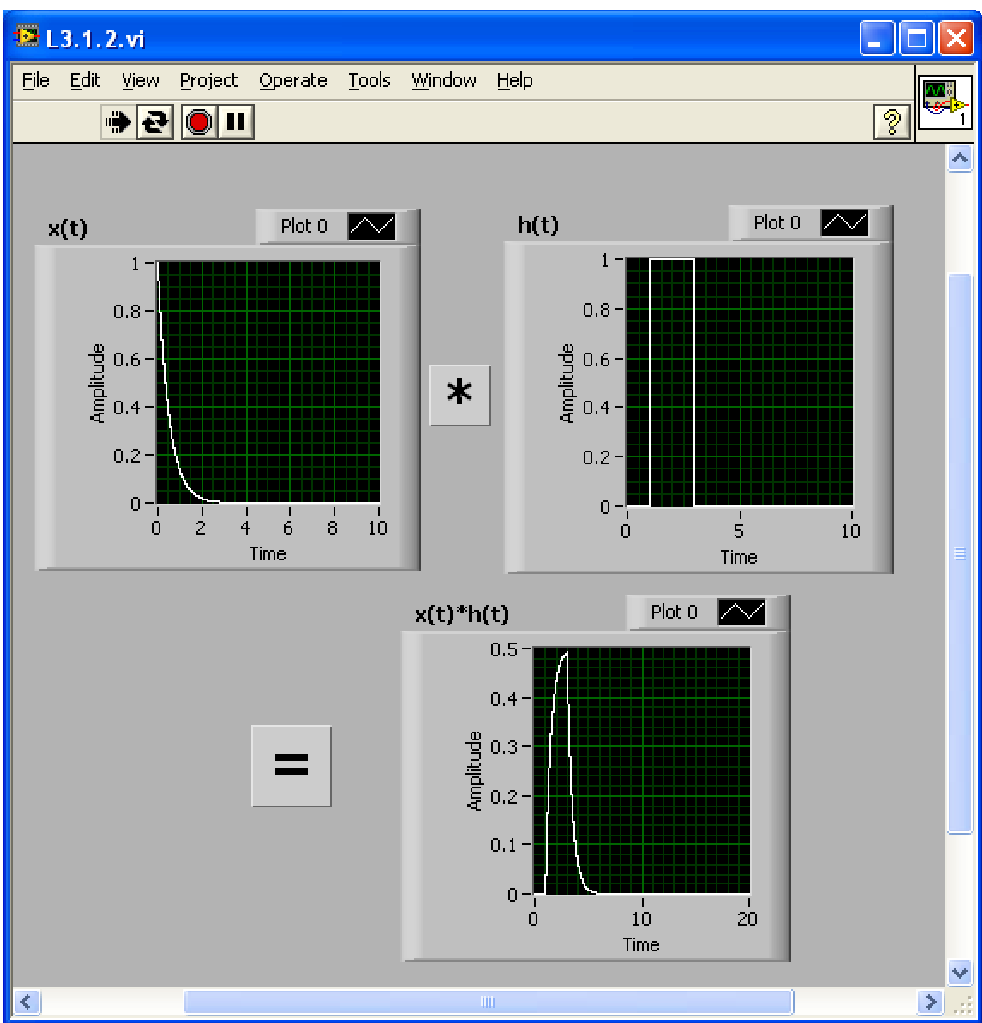
$$\text{rect}(t) = \begin{cases} 1 & -0.5 \leq t < 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Let $\Delta = 0.01$. [link] shows the block diagram for this second convolution example. Again, the .m file textual code is placed inside a LabVIEW MathScript node with the appropriate inputs and outputs.



Block Diagram for the Convolution of Two Signals

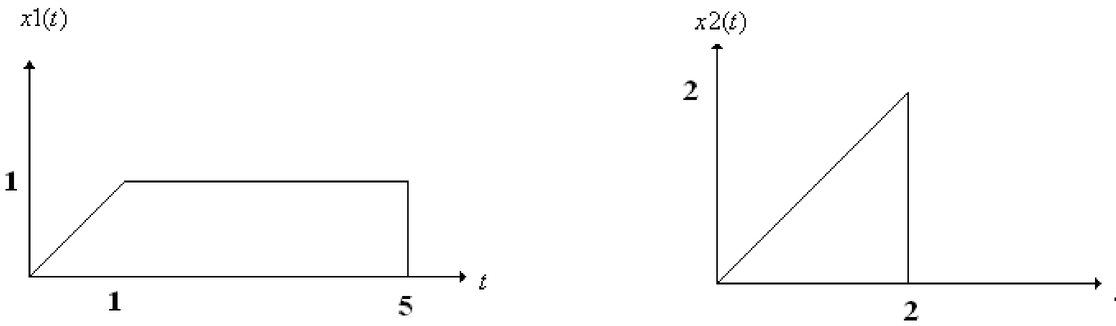
[link] illustrates the corresponding front panel where $x(t)$, $h(t)$ and $x(t) * h(t)$ are plotted in different graphs. Convolution $(*)$ and equal $(=)$signs are placed between the graphs using the LabVIEW function `Decorations`.



Front Panel for the Convolution of Two Signals

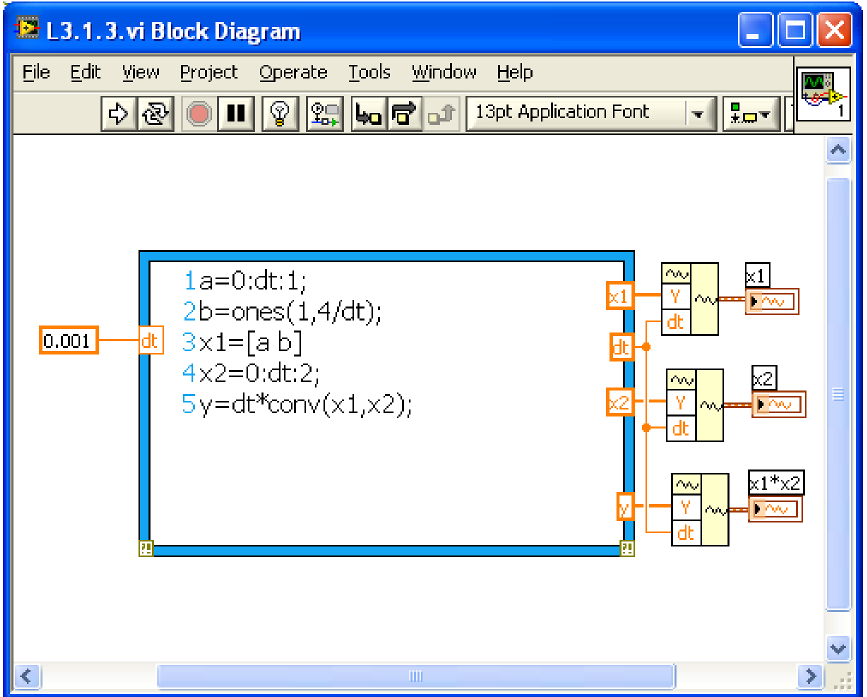
Convolution Example 3

In this third example, compute the convolution of the signals shown in [link].

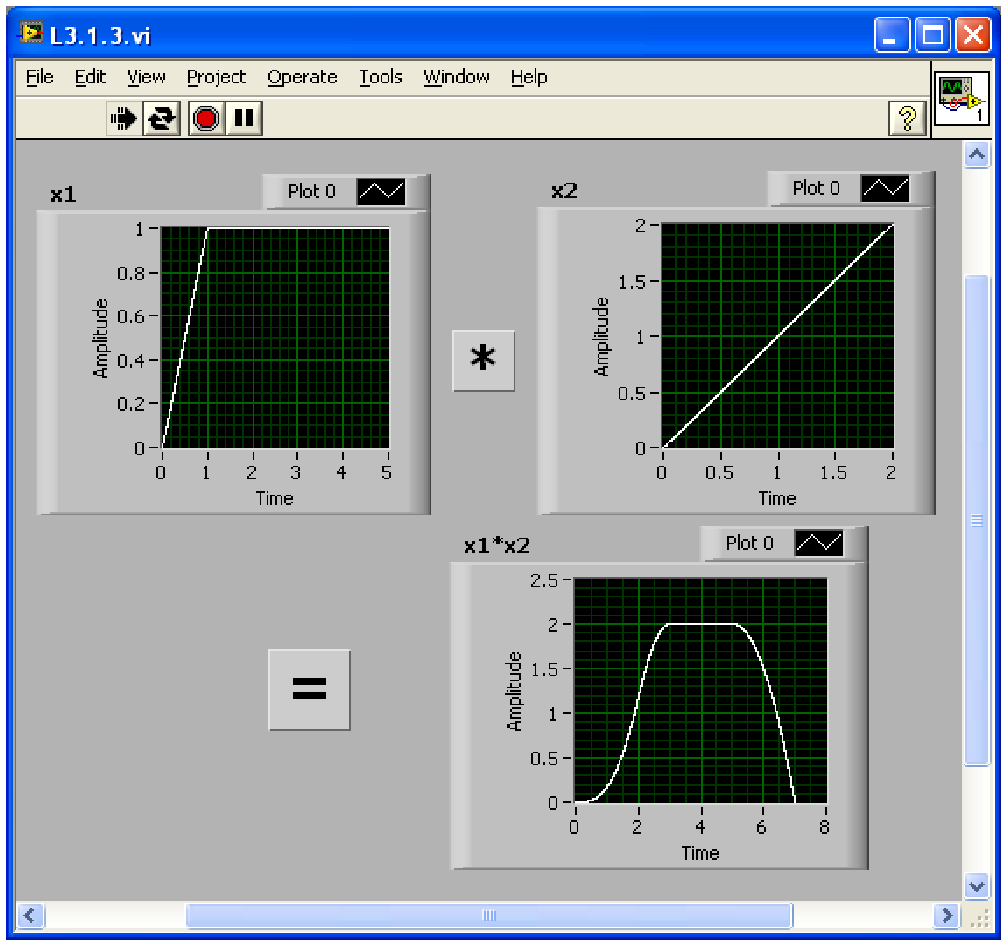


Signals x1(t) and x2(t)

[link] shows the block diagram for this third convolution example and [link] the corresponding front panel. The signals $x1(t)$, $x2(t)$ and $x1(t) * x2(t)$ are displayed in different graphs.



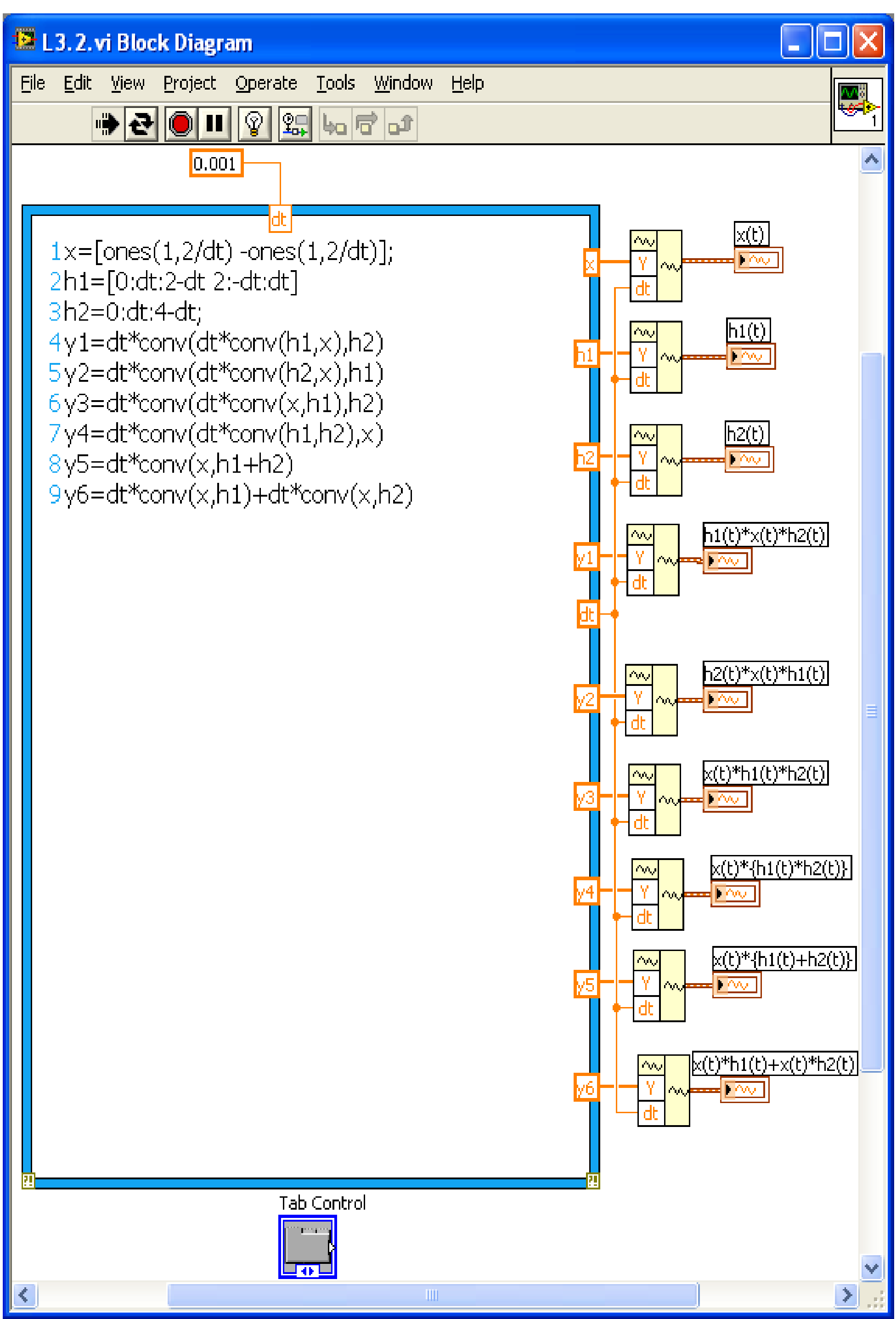
Block Diagram for the Convolution of Two Signals



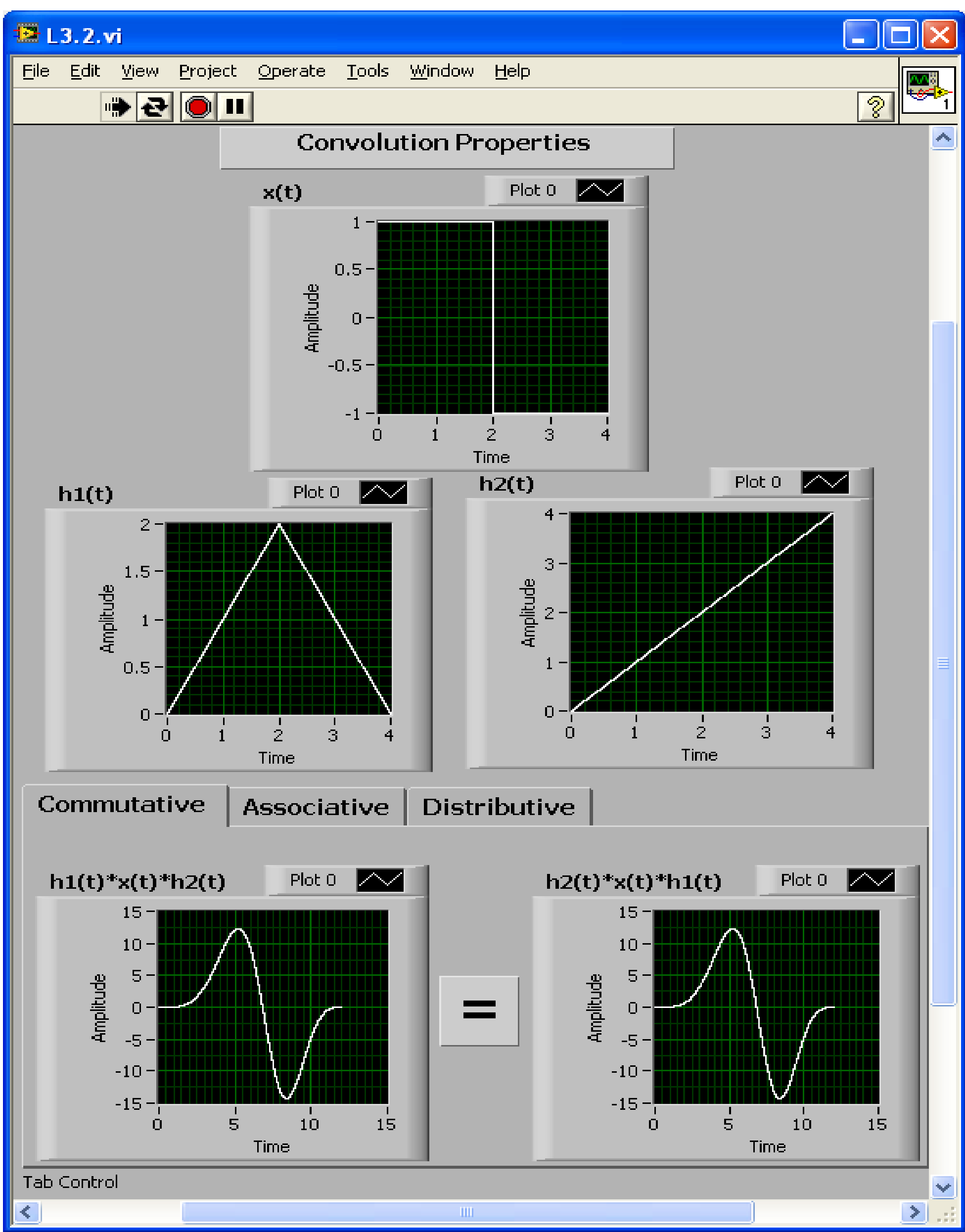
Front Panel for the Convolution of Two Signals

Convolution Properties

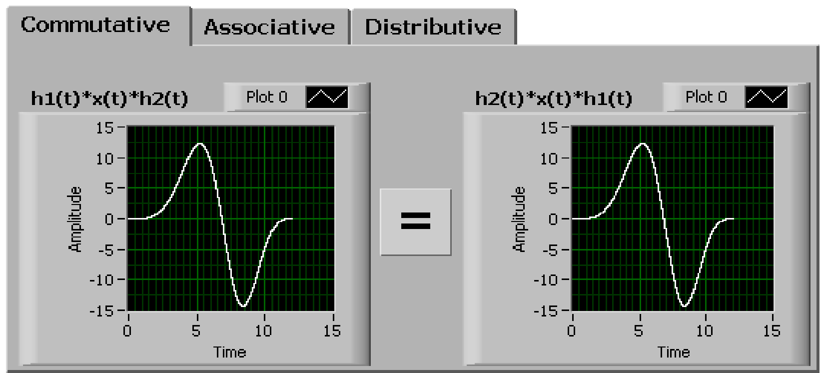
In this part, examine the properties of convolution. [link] shows the block diagram to examine the properties and [link] and [link] the corresponding front panel. Both sides of equations are plotted in this front panel to verify the convolution properties. To display different convolution properties within a limited screen area, use a **Tab Control (Controls →Modern→Containers→Tab Control)** in the front panel.

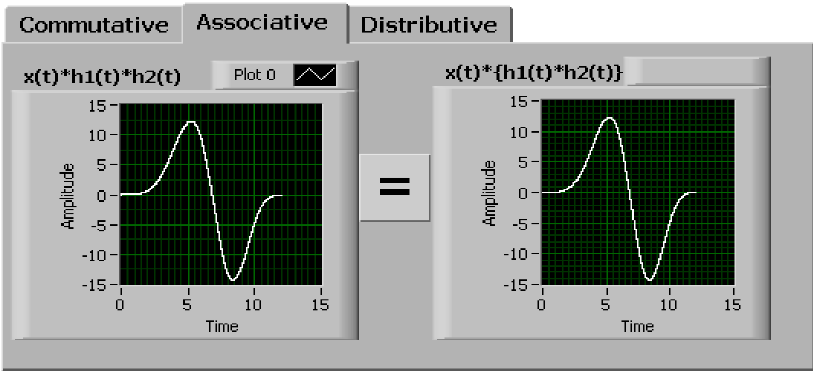


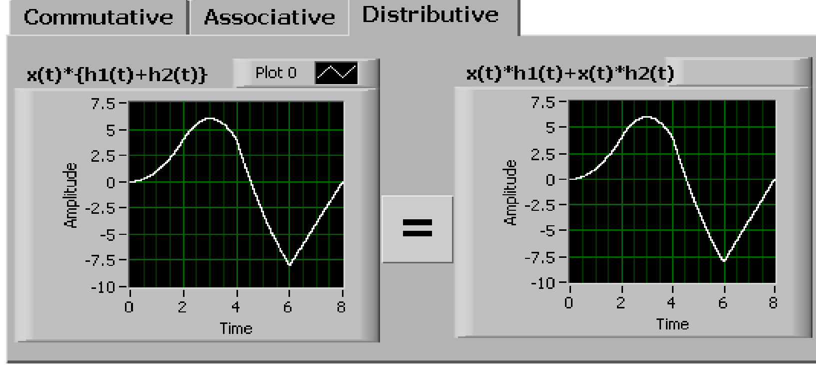
Front Panel of Convolution Properties



Block Diagram of Convolution Properties



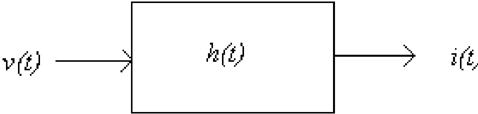




Tabs Showing Convolution Properties

Linear Circuit Analysis Using Convolution

In this part, let us consider an application of convolution in analyzing RLC circuits to gain a better understanding of the convolution concept. A linear circuit denotes a linear system, which can be represented with its impulse response $h(t)$, that is, its response to a unit impulse input. The input to such a system can be considered to be a voltage $v(t)$and the output to be the circuit current $i(t)$. See [link].



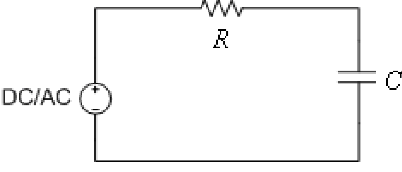
Impulse Response Representation
of a Linear Circuit

For a simple RC series circuit shown in [link], the impulse response is given by [link] ,
Equation:

$$h(t) = \frac{1}{\text{RC}}\exp(-\frac{1}{\text{RC}}t)$$

which can be obtained for any specified values of R and C. When an input voltage $v(t)$ (either DC or AC) is applied to the system, the circuit current $i(t)$ can be obtained by simply convolving the system impulse response with the input voltage, that is
Equation:

$$i(t) = h(t) * v(t)$$

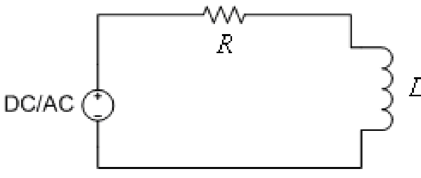


RC Circuit

Similarly, for the simple RL series circuit shown in [link], the impulse response is given by [link] ,
Equation:

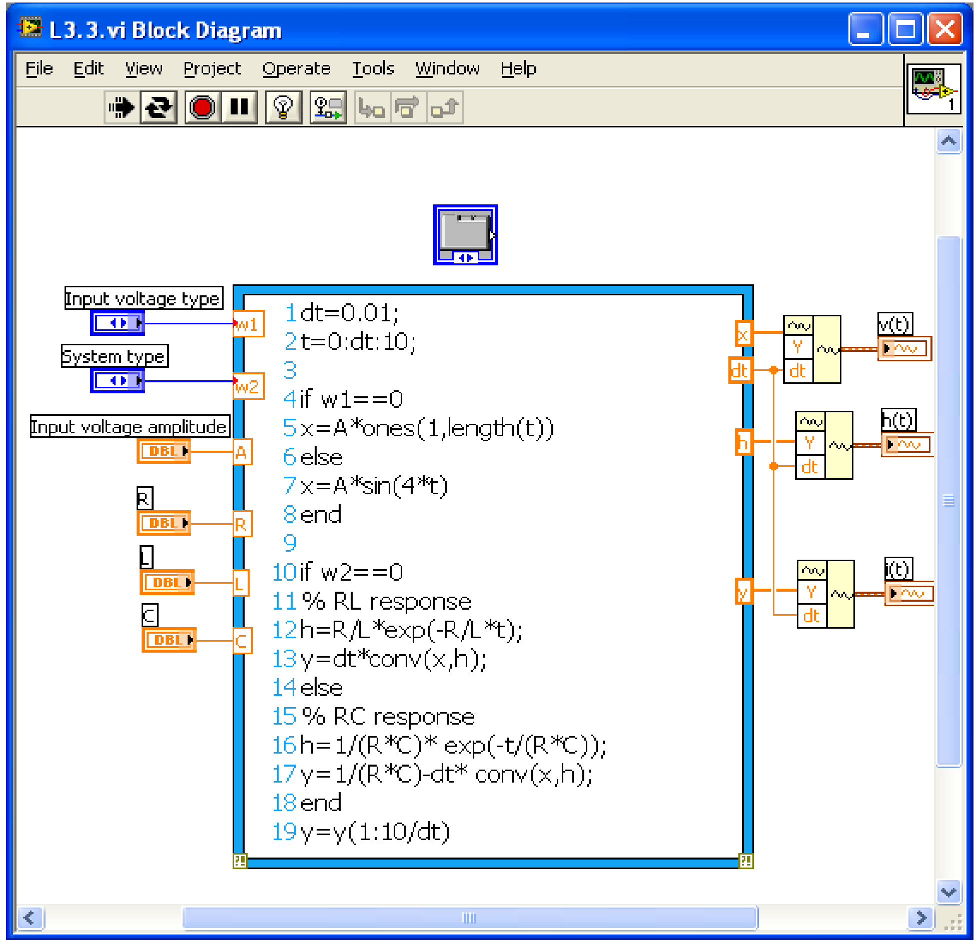
$$h(t) = \frac{\text{R}}{L}\exp(-\frac{R}{L}t)$$

When an input voltage $v(t)$ is applied to the system, the circuit current $i(t)$ can be obtained by computing the convolution integral.

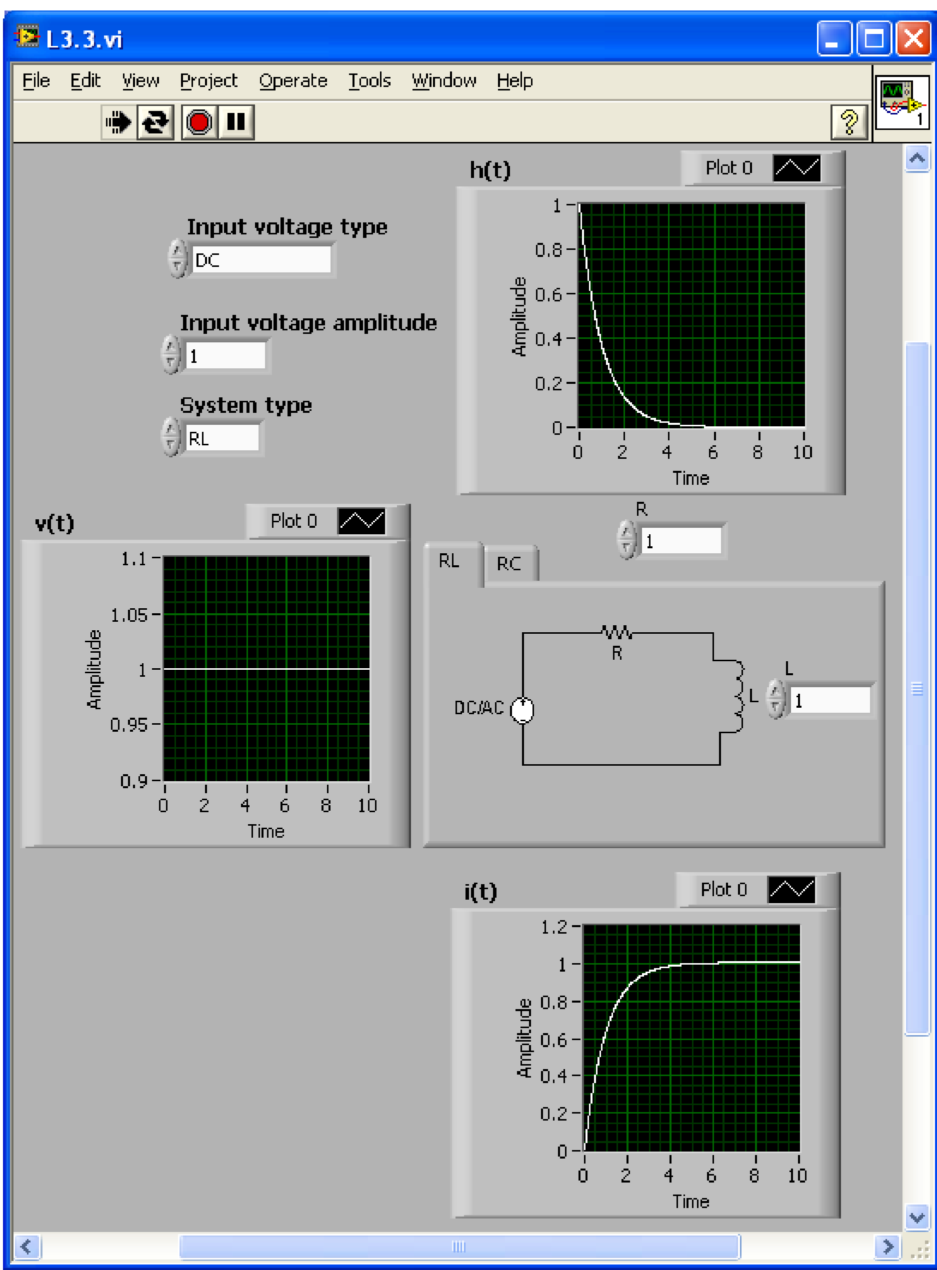


RL Circuit

[link] shows the block diagram of this linear system and [link] the corresponding front panel. From the front panel, one can control the system type (RL or RC), input voltage type (DC or AC) and input voltage amplitude. One can also observe the system response by changing R, L and C values. Three graphs are used to display the input voltage $v(t)$, impulse response of the circuit $h(t)$ and circuit current $i(t)$.



Block Diagram of the Linear Circuit Application



Front Panel of the Linear Circuit Application

Lab Exercises

Exercise:

Problem: Echo Cancellation

In this exercise, consider the problem of removing an echo from a recording of a speech signal. The LabVIEW MathScript function `sound()` or the function **Play Waveform** in LabVIEW can be used to play back the speech recording. To begin, load the .m file echo_1.wav provided on the book website by using the function `wavread('filename')`. This speech file was recorded at the sampling rate of 8 kHz, which can be played back through the computer speakers by typing

```
>> sound(y)
```

You should be able to hear the sound with an echo. If the LabVIEW function **Play Waveform(Functions → Programming → Graphics & Sound → Sound→ Output→ Play Waveform)** is used to play the sound, you first need to build a waveform based on the loaded data and the time interval $\mathrm{dt} = 1/8000$ because this speech was recorded using an 8 kHz sampling rate. Connect the waveform to the function **Play Waveform**.

An echo is produced when the signal (speech, in this case) is reflected off a non-absorbing surface like a wall. What is heard is the original signal superimposed on the signal reflected off the wall (echo). Because the speech is partially absorbed by the wall, it decreases in amplitude. It is also delayed. The echoed signal can be modeled as $\mathrm{ax}(t-\tau)$where $a<1$ and τ denotes the echo delay. Thus, one can represent the speech signal plus the echoed signal as [7]

Equation:

$$y(t) = x(t) + \mathrm{ax}(t-\tau)$$

What is heard is $y(t)$. In many applications, it is important to recover $x(t)$ – the original, echo-free signal – from $y(t)$.

Method 1

In this method, remove the echo using deconvolution. Rewrite Equation (7) as follows [7]:

Equation:

$$y[n\Delta] = x[n\Delta] + \mathrm{ax}[(n-N)\Delta] = x[n\Delta] * (\delta[n\Delta] + a\delta[n-N]\Delta) = x[n\Delta] * h[n\Delta]$$

The echoed signal is the convolution of the original signal $x(n\Delta)$ and the signal $h(n\Delta)$. Use the LabVIEW MathScript function `deconv(y,h)` to recover the original signal.

Method 2

An alternative way of removing the echo is to run the echoed signal through the following system:
Equation:

$$z[n\Delta] = y[n\Delta] - \mathrm{az}[(n - N)\Delta]$$

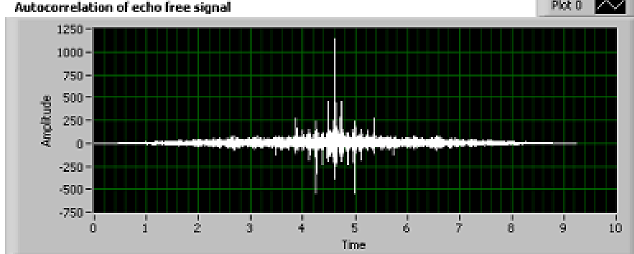
Assume that $z[n\Delta] = 0$for $n < 0$. Implement the above system for different values of a and N.

Display and play back the echoed signal and the echo-free signal using both of the above methods. Specify the parameters $a$and Nas controls. Try to measure the proper values of $a$and $N$by the autocorrelation method described below.

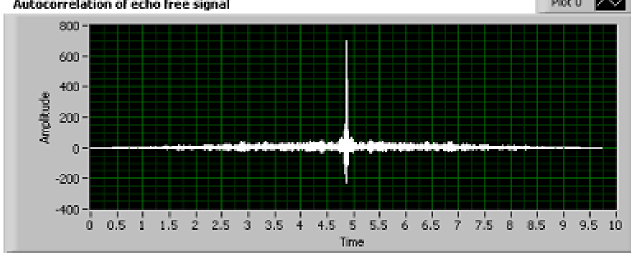
The autocorrelation of a signal can be described by the convolution of a signal with its mirror. That is,
Equation:

$$R_{\mathrm{xx}}[n] = x[n] * x[-n]$$

Use the autocorrelation of the output signal (echo-free signal) to estimate the delay time (N) and the amplitude of the echo (a). For different values of $N$and a, observe the autocorrelation output. To have an echo-free signal, the side lobes of the autocorrelation should be quite low, as shown in [link].



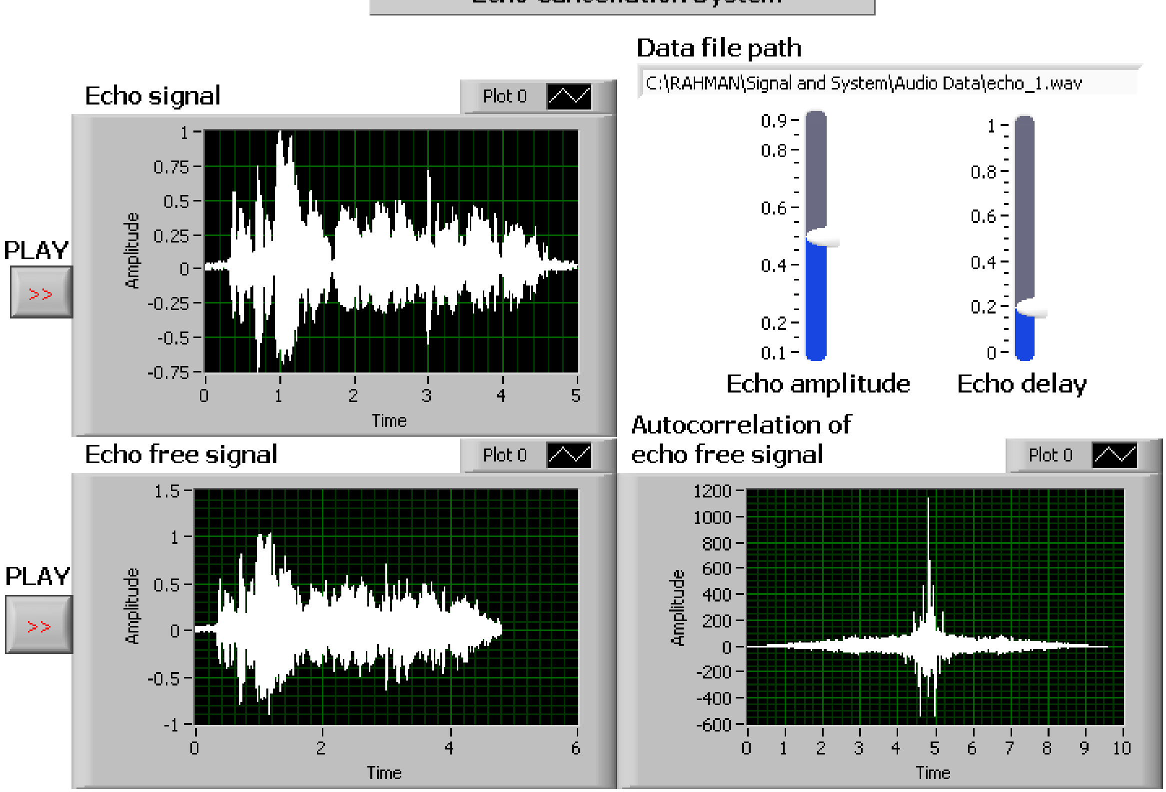
(a)



(b)

Autocorrelation Function of a Signal: (a) Echo Is Not Removed Completely; (b) Echo Is Removed

[link] shows a typical front panel for this exercise. It is not necessary to obtain the same front panel but there should be controls for a and N as well as graphs to observe the echoed signal, echo-free signal and autocorrelation function of the echo-free signal.



Front Panel for the Echo Cancellation System

Solution:

Insert Solution Text Here

Exercise:

Problem: Noise Reduction Using Mean Filtering

The idea of mean filtering is simply to replace each value in a signal with the mean (average) value of its neighbors. A mean filter is widely used for noise reduction.

Start by adding some random noise to a signal (use the file echo_1.wav or any other speech data file). Then, use mean filtering to reduce the introduced noise. More specifically, take the following steps:

1. Normalize the signal values in the range [0 1].
2. Add random noise to the signal by using the function `randn`. Set the noise level as a control.
3. Convolve the noise-added signal with a mean filter. This filter can be designed by taking an odd number of ones and dividing by the size. For example, a 1×3 size mean filter is given by [1/3 1/3 1/3] and a 1×5size mean filter by [1/5 1/5 1/5 1/5 1/5]. Set the size of the mean filter as an odd number control (3, 5 or 7, for example).

Solution:

Insert Solution Text Here

Exercise:

Problem: Impulse Noise Reduction Using Median Filtering

A median filter is a non-linear filter that replaces a data value with the median of the values within a neighboring window. For example, the median value for this data stream [2 5 3 11 4] is 4. This type of filter is often used to remove impulse noise. Use the file echo_1.wav or any other speech data file and take the following steps:

1. Normalize the signal values in the range [0 1].
2. Randomly add impulse noise to the signal by using the LabVIEW MathScript function `randperm`. Set the noise density as a control.
3. Find the median values of neighboring data using the function `median` and replace the original value with the median value. Set the number of neighboring values as an odd number control (3, 5 or 7, for example).

Solution:

Insert Solution Text Here

Fourier Series

Background

A periodic signal $x(t)$can be expressed by an exponential Fourier series as follows:
Equation:

$$x(t) = \sum_{n=-\infty}^{\infty} c_n e^{j\frac{2\pi nt}{T}}$$

where T indicates the period of the signal and c_n’s are called Fourier series coefficients, which, in general, are complex. Obtain these coefficients by performing the following integration
Equation:

$$c_n = \frac{1}{T}\int_T x(t)e^{-j\frac{2\pi nt}{T}}\mathrm{dt}$$

which possesses the following symmetry properties
Equation:

$$\mid c_{-n} \mid = \mid c_n \mid$$

Equation:

$$\angle c_{-n} = -\angle c_n$$

where the symbol $\mid . \mid$ denotes magnitude and $\angle$ phase. Magnitudes of the coefficients possess even symmetry and their phases odd symmetry.

A periodic signal $x(t)$ can also be represented by a trigonometric Fourier series as follows:
Equation:

$$x(t) = a_0 + \sum_{n=1}^{\infty} a_n \cos(\frac{2\pi \mathrm{nt}}{T}) + b_n \sin(\frac{2\pi \mathrm{nt}}{T})$$

where

Equation:

$$a_0 = \frac{1}{T} \int_T x(t) \mathrm{dt}$$

Equation:

$$a_n = \frac{2}{T} \int_T x(t) \cos(\frac{2\pi \mathrm{nt}}{T}) \mathrm{dt}$$

Equation:

$$b_n = \frac{2}{T} \int_T x(t) \sin(\frac{2\pi \mathrm{nt}}{T}) \mathrm{dt}$$

The relationships between the trigonometric series and the exponential series coefficients are given by

Equation:

$$a_0 = c_0$$

Equation:

$$a_n = 2 \mathrm{Re}\{c_n\}$$

Equation:

$$b_n = -2 \mathrm{Im}\{c_n\}$$

Equation:

$$c_n = \frac{1}{2}(a_n - \mathrm{jb}_n)$$

where Re and Imdenote the real and imaginary parts, respectively.

According to the Parseval's theorem, the average power in the signal $x(t)$ is related to the Fourier series coefficients c_n's, as indicated below

Equation:

$$\frac{1}{T}\int_T |\, x(t)\, |^2 \mathrm{dt} = \sum_{n=-\infty}^{\infty} |\, c_n\, |^2$$

More theoretical details of Fourier series are available in signals and systems textbooks [link] - [link] .

Fourier Series Numerical Computation

Fourier series coefficients are often computed numerically – in particular, when an analytic expression for $x(t)$ is not available or the integration in [link] - [link] is difficult to perform. By approximating the integrals in [link] - [link] with a summation of rectangular strips, each of width Δt, one can write

Equation:

$$a_0 = \frac{1}{M}\sum_{m=1}^{M} x(m\Delta t)$$

Equation:

$$a_n = \frac{2}{M}\sum_{m=1}^{M} x(m\Delta t)\cos(\frac{2\pi\mathrm{mn}}{M})$$

Equation:

$$b_n = \frac{2}{M} \sum_{m=1}^{M} x(m\Delta t) \sin(\frac{2\pi mn}{M})$$

where $x(m\Delta t)$ are M equally spaced data points representing $x(t)$ over a single period T, and Δt denotes the interval between data points such that $\Delta t = \frac{T}{M}$

Similarly, by approximating the integrals in [link] with a summation of rectangular strips, each of width Δt, one can write
Equation:

$$c_n = \frac{1}{M} \sum_{m=M}^{M} x(m\Delta t) \exp(\frac{j2\pi mn}{M})$$

Lab 4: Fourier Series and Its Applications

In this lab, we examine the representation of periodic signals based on Fourier series. Periodic signals can be represented by a linear combination of an infinite sum of sine waves, as expressed by the trigonometric Fourier series representation.Periodic signals can also be represented by an infinite sum of harmonically related complex exponentials, as expressed by the exponential Fourier series representation. In this lab, we analyze both of these series representations. In particular, we focus on how to compute Fourier series coefficients numerically.

Fourier Series Signal Decomposition and Reconstruction

This example helps one to gain an understanding of Fourier series decomposition and reconstruction for periodic signals. The first step involves estimating $x(m\Delta t)$ which is a numerical approximation of the analog input signal. Though programming environments deploy discrete values internally, we can obtain a close analog approximation of a continuous-time signal by using a very small Δt. That is to say, for all practical purposes, when Δt is taken to be very small, we get the analog representation or simulation of the signal. In this example, create four input signals using the listed LabVIEW MathScript functions in Table 1.

Waveform type	**LabVIEW MathScript function**
Square wave	`square(T)`, T denotes period
Triangular wave	`sawtooth(T,Width)`, Width=0.5
Sawtooth wave	`sawtooth(T,Width)`, Width=0
Half wave rectified sine wave	$\begin{cases} \sin(2*\text{pi}*f*t) & \text{for} \quad 0 \le t < T/2 \\ 0 & \text{for} \quad T/2 \le t < T \end{cases}$, $f = 1/T$denotes frequencyHalf period is sine wave and the other half is made zero

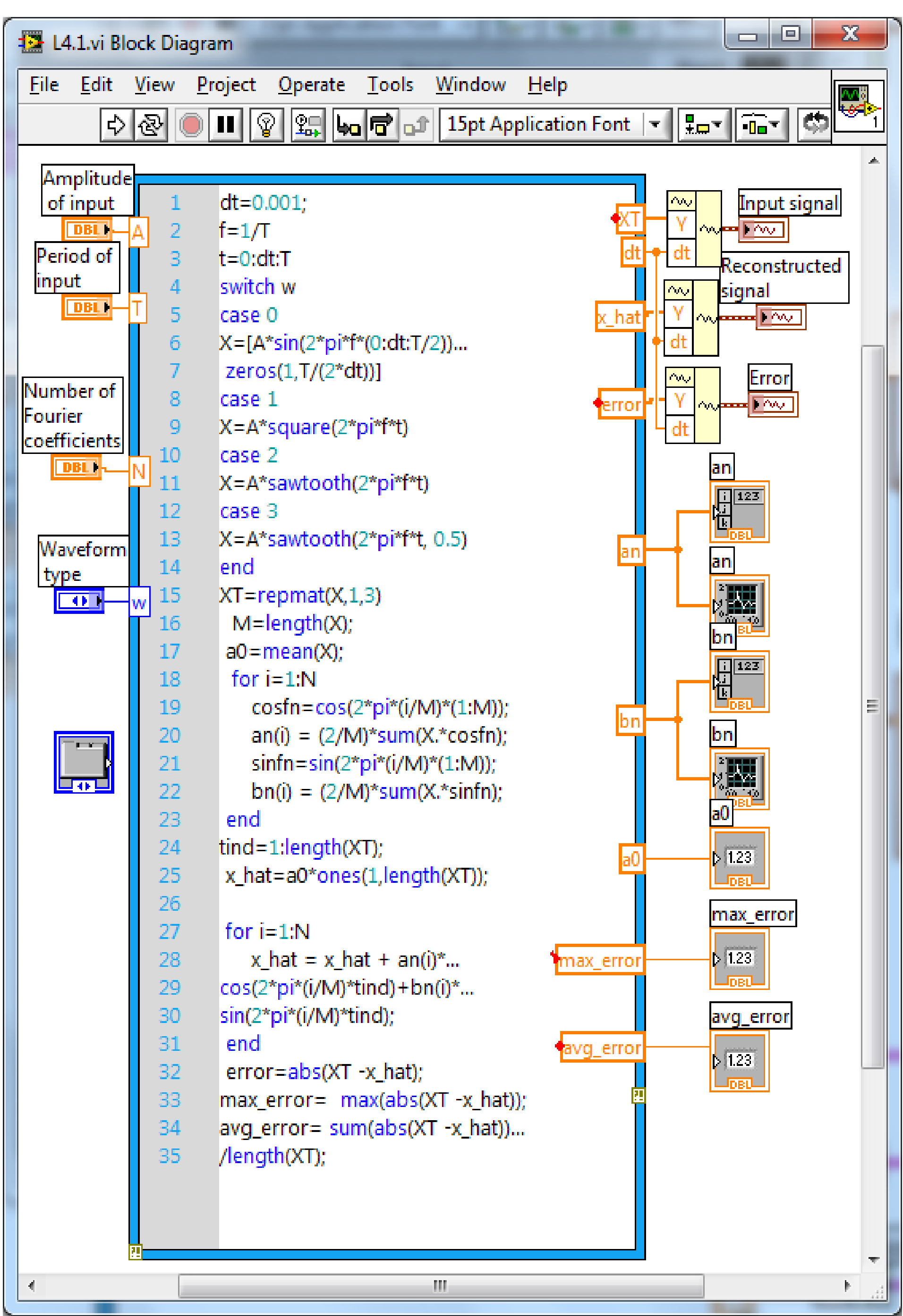
LabVIEW MathScript Functions for Generating Various Waveforms or Signals

Use a `switch` structure to select different types of input waveforms. Set the switch parameter w as the input and connect it to an **Enum Control(Controls → Modern → Ring & Enum → Enum)**. Edit the Enum Control items to include all the waveform types.

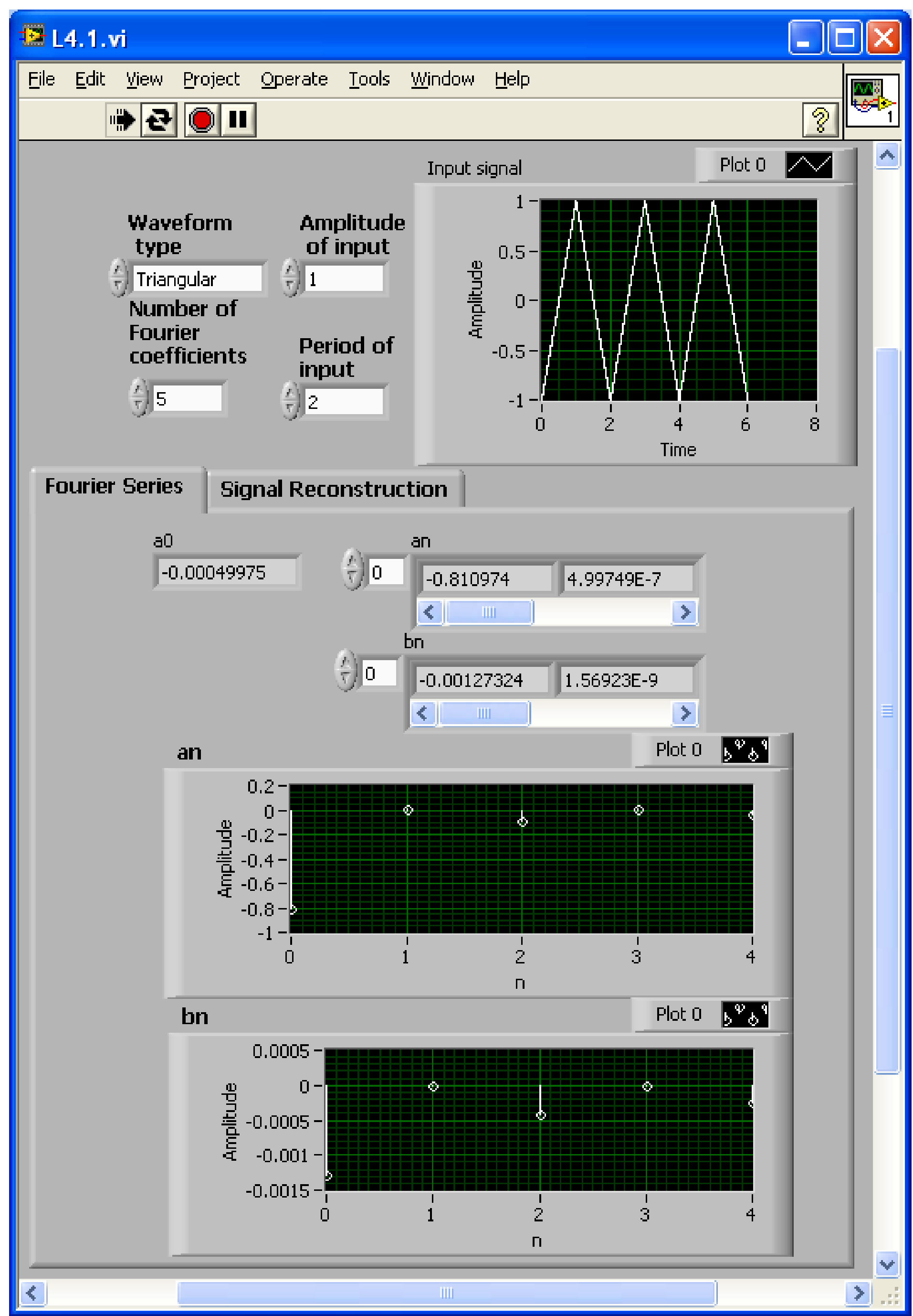
Set Amplitude of input (A), Period of input (T) and Number of Fourier coefficients (N) as control parameters. Determine Fourier coefficients a_0,$a_n$and b_n and reconstruct the signal from its Fourier coefficients using equations provided in Chapter 4. Determine the error between the input and the reconstructed signal by simply taking the absolute values of $x(t)-\hat{x}(t)$via the LabVIEW MathScript function `abs`. Finally, determine the maximum and average errors by using the functions `max` and `sum`. [link] shows the completed block diagram of the Fourier series signal decomposition and reconstruction system.

Display the input signal using a waveform graph. Before displaying the graph, configure it using the function **Build Waveform(Functions → Programming → Waveforms → Build Waveforms)**. Also display the Fourier coefficients, reconstructed signal and error in the waveform graph, and place several numerical indicators to show the values of the Fourier coefficients, maximum error and average error.

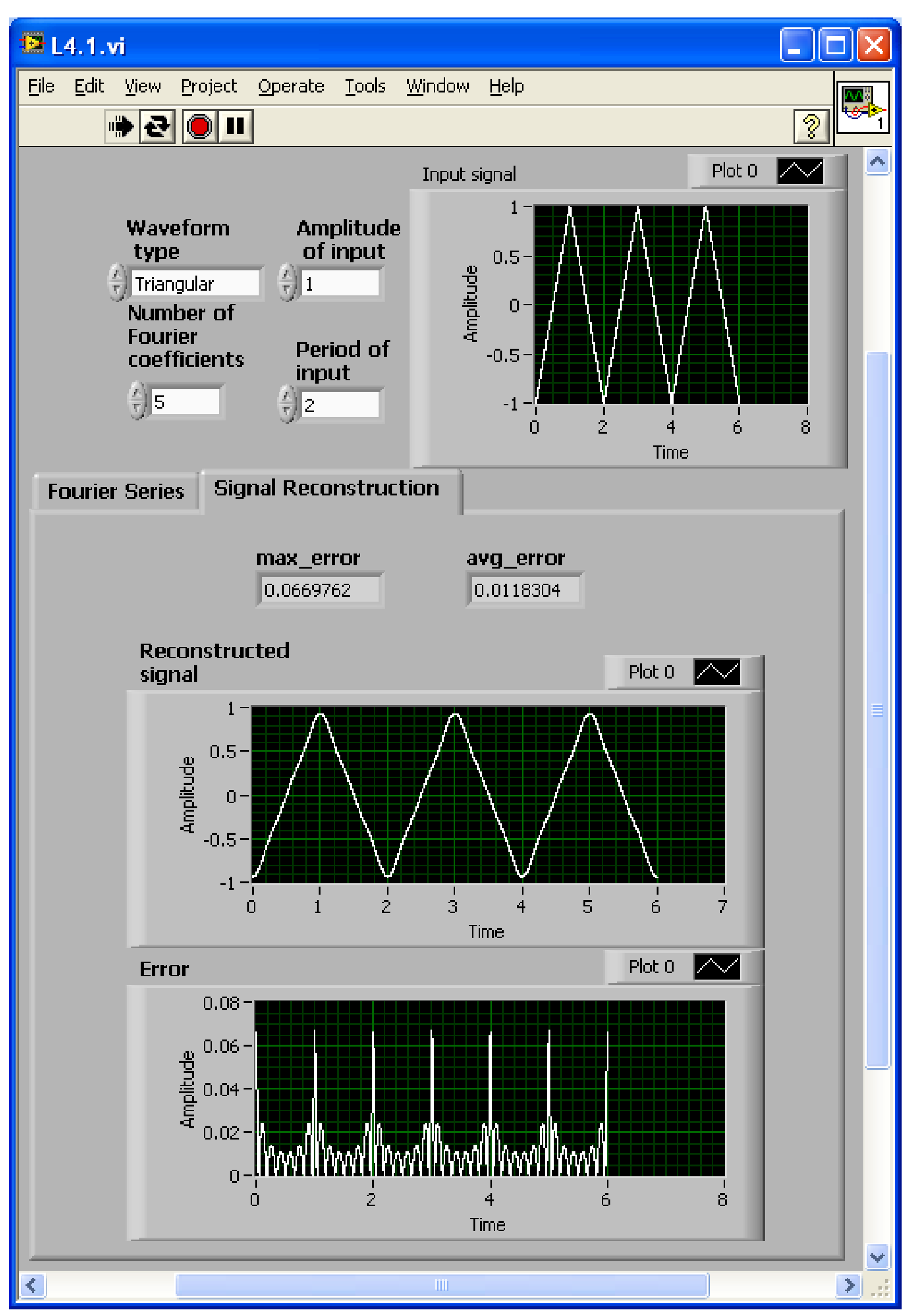
[link] and [link] illustrate the front panel of the Fourier series signal decomposition and reconstruction system, respectively. To display all the outputs within a limited screen area, use a **Tab Control(Controls → Modern → Containers → Tab Control)** in the front panel. Here the outputs are arranged in two different tabs: Fourier Series and Signal Reconstruction.



Block Diagram of Fourier Series Signal Decomposition and Reconstruction Example



Front Panel of Fourier Series Signal Decomposition and Reconstruction Example (Fourier Series Tab)

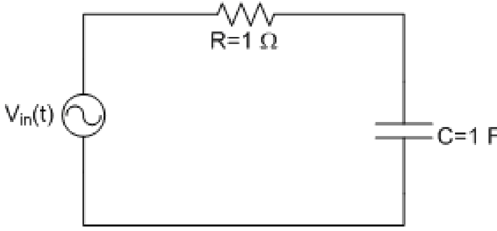


Front Panel of Fourier Series Signal Decomposition and Reconstruction Example (Signal Reconstruction Tab)

Linear Circuit Analysis Using Trigonometric Fourier Series

In this example, let us perform electrical circuit analysis using the trigonometric Fourier series. The ability to decompose any periodic signal into a number of sine waves makes the Fourier series a powerful tool in electrical circuit analysis. The response of a circuit component when a sinusoidal input is applied to its terminals is well-known in circuit analysis. Thus, to obtain the response to any periodic signal, one can decompose the signal into sine waves and perform a linear superposition of the sine waves.

Consider a simple RC circuit excited by a periodic input signal as shown in [link].



RC Series Circuit with Periodic Input Voltage

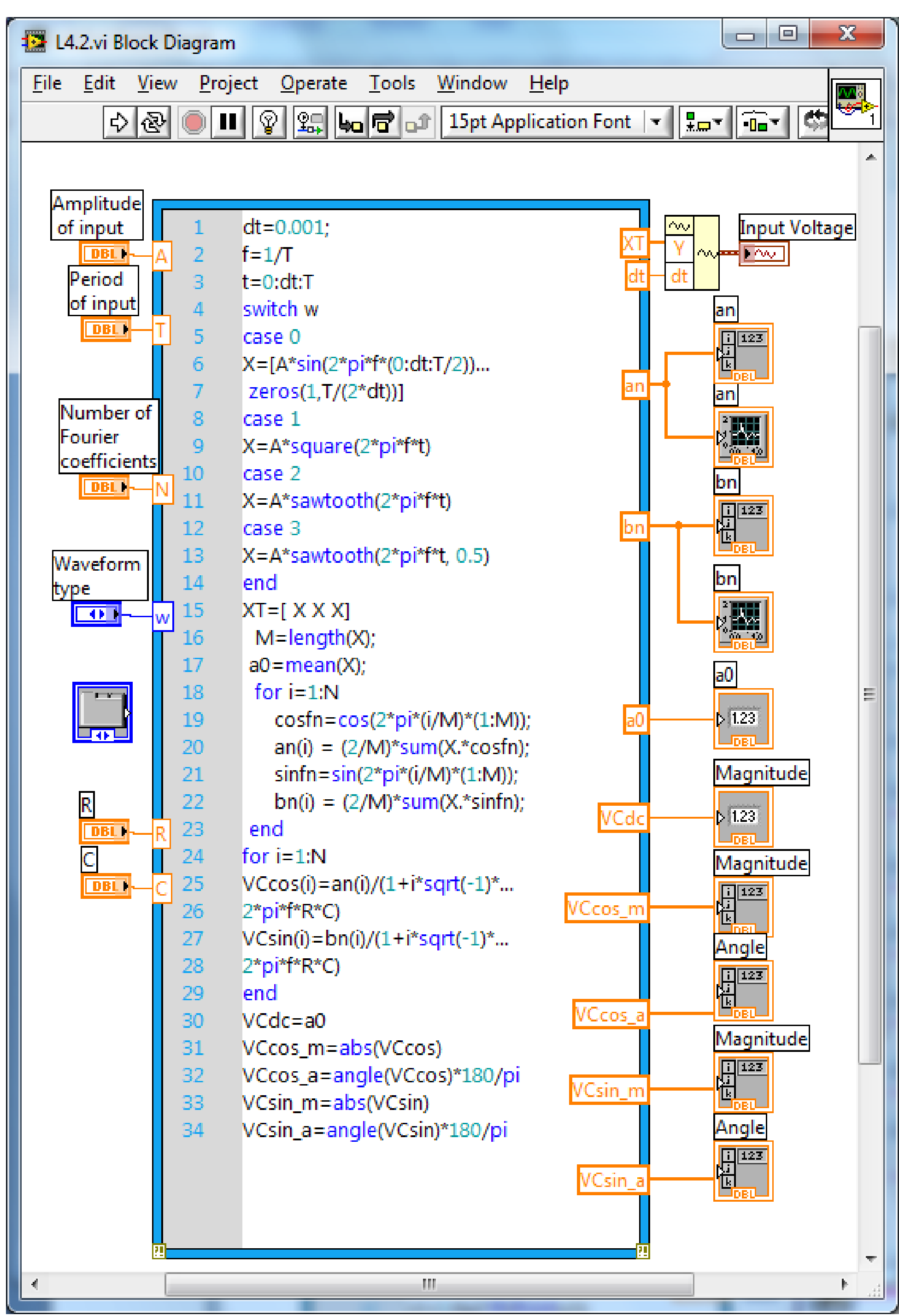
The block diagram of this linear circuit or system is shown in [link]. Determine the Fourier series coefficients of the input voltage signal as discussed in the previous example. Because the Fourier series involves the sum of sinusoids, phasor analysis can be used to obtain the output voltage (v_c). Let n represent the number of terms in the Fourier series. By using the voltage divider rule, the output voltage (v_c) can be expressed as [link],

Equation:

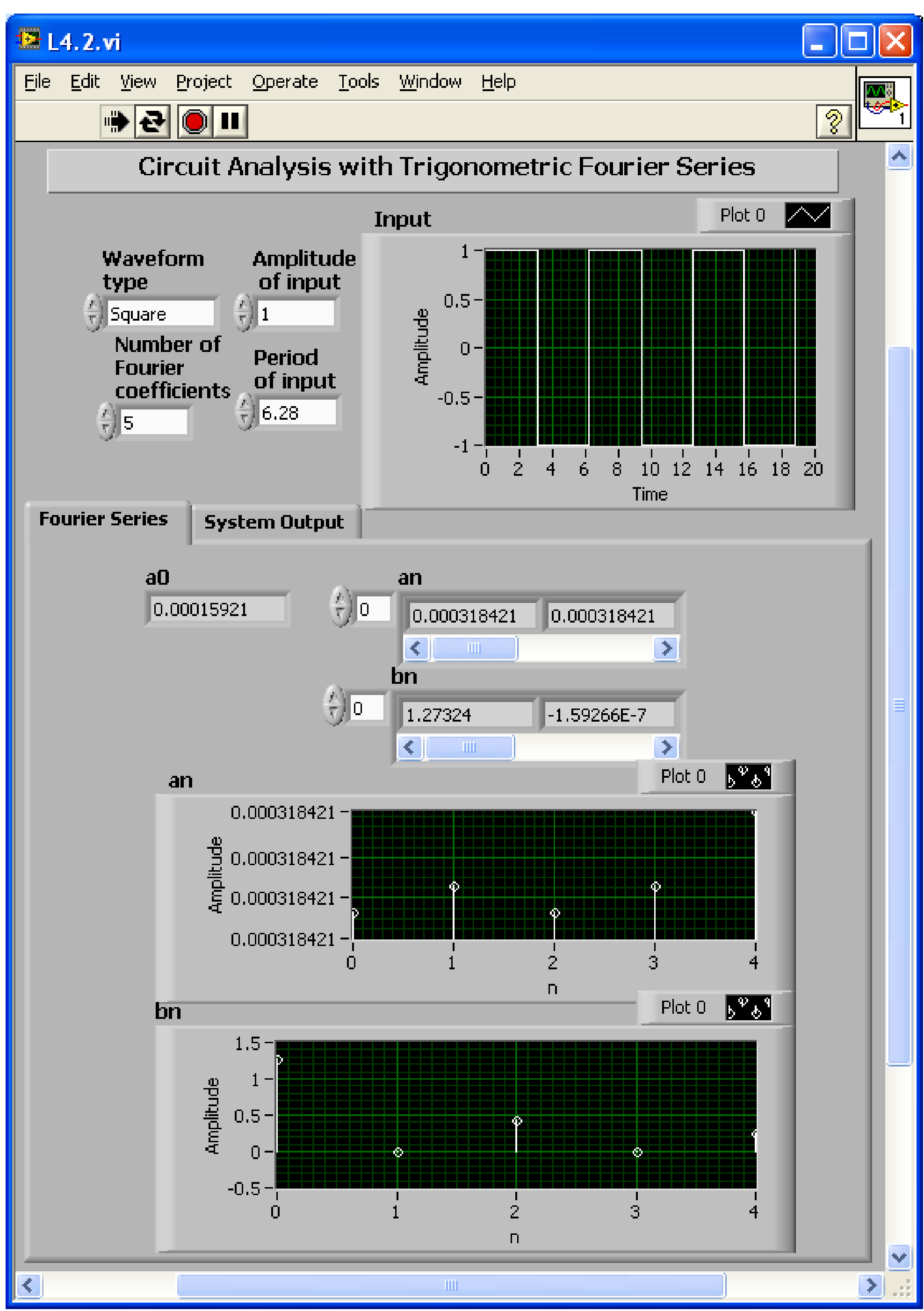
$$v_{c_n} = \frac{1/(\mathrm{j}n\omega C)}{R + 1/(\mathrm{j}n\omega C)} v_{\mathrm{in}_n}$$

Because the sine and cosine components of the input voltage are known, one can easily determine the output by adding the individual output components because the circuit is linear. Determine each output voltage component by using [link].

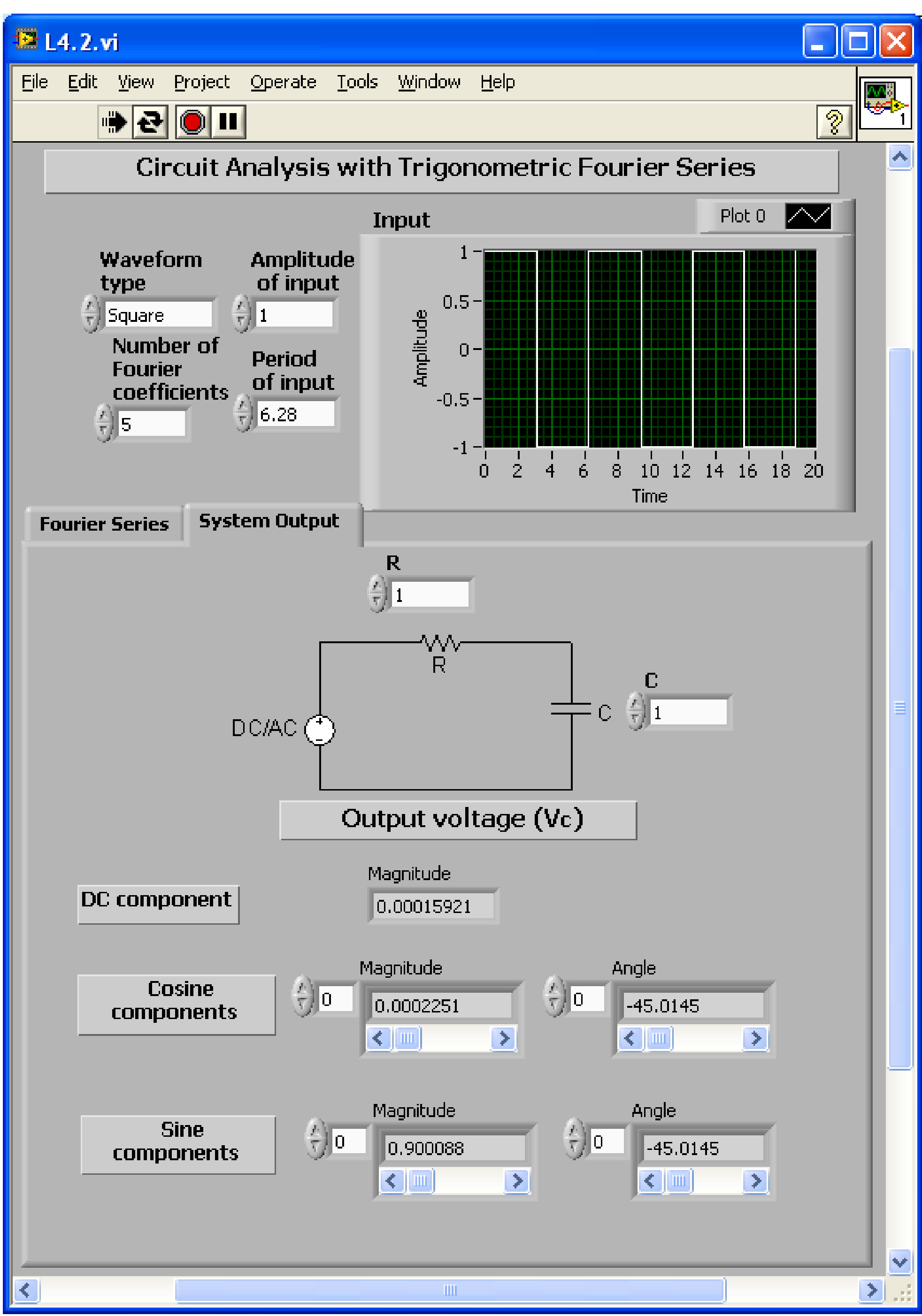
[link] and [link] show the front panel of this system for its two tabs. The magnitude and phase of the sine and cosine components are shown in the front panel separately. Furthermore, the tab control is used to show the Fourier series and system output separately.



Block Diagram of Circuit Analysis with Trigonometric Fourier Series



Front Panel of Circuit Analysis with Trigonometric Fourier Series
(Fourier Series Tab)



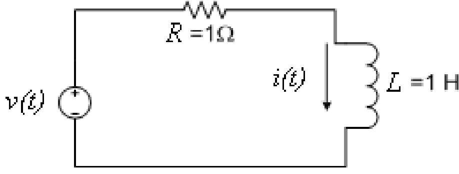
Front Panel of Circuit Analysis with Trigonometric Fourier Series (System Output Tab)

Lab Exercises

Exercise:

Problem: RL Circuit Analysis

Build a hybrid VI to analyze the RL circuit shown in [link] using Fourier series.



RL Series Circuit with Periodic Input Voltage

The input voltage for the circuit is to be either a square wave or a triangular wave with a period T=2 seconds.

Compute and display the following:

1. The Fourier series coefficients of the input voltage $v(t)$,
2. the current $i(t)$,
3. the RMS (root mean square) value of $v(t)$ using (i) the original waveform and (ii) its Fourier series coefficients (compare the outcomes),
4. the average power P_{av}delivered by the source.

Hints:

RMS Value

The RMS value of a periodic function $v(t)$with period $T$is given by
Equation:

$$V_{\text{RMS}} = \sqrt{\frac{1}{T}\int_{T} v^2 \text{dt}}$$

The RMS value of a waveform consisting of sinusoids with different frequencies is equal to the square root of the sum of the squares of the RMS value of each sinusoid. If a waveform is represented by the following Fourier series
Equation:

$$v(t) = V_0 + V_1 \sin(\omega_1 t \pm \varphi_1) + V_2 \sin(\omega_2 t \pm \varphi_2) + \ldots + V_N \sin(\omega_N t \pm \varphi_N)$$

then, the RMS value V_{RMS} is given by

Equation:

$$V_{\text{RMS}} = \sqrt{V_{0^2} + \left(\frac{V_1}{\sqrt{2}}\right)^2 + \left(\frac{V_2}{\sqrt{2}}\right)^2 + \ldots + \left(\frac{V_N}{\sqrt{2}}\right)^2}$$

Average power

The average power of the Fourier series can be expressed as

Equation:

$$P_{\text{av}} = V_0 I_0 + V_{1\text{RMS}} I_{1\text{RMS}} \cos\varphi_1 + V_{2\text{RMS}} I_{2\text{RMS}} \cos\varphi_2 + \ldots$$

Solution:

Insert Solution Text Here

Exercise:

Problem: Doppler Effect

The Doppler effect denotes the change in frequency and wavelength of a wave as perceived by an observer moving relative to the wave source. The Doppler effect can be demonstrated via time scaling of Fourier series. The observer hears the siren of an approaching emergency vehicle with different amplitudes and frequencies as compared to the original signal. As the vehicle passes by, the observer hears another amplitude and frequency. The reason for the amplitude change (increased loudness) is because of the proximity of the vehicle. The closer it is, the louder it gets. The reason for frequency (pitch) change is due to the Doppler effect. As the vehicle approaches, each successive compression of the air caused by the siren occurs a little closer than the last one, and the opposite happens when the vehicle passes by. The result is the scaling of the original signal in the time domain, which changes its frequency. When the vehicle approaches, the scaling factor is greater than 1, resulting in a higher frequency, and, when it passes by, the scaling factor is less than 1, resulting in a lower frequency. More theoretical aspects of this phenomenon are covered in reference [link].

Define the original siren signal as $x(t)$. When the vehicle approaches, one can describe the signal by

Equation:

$$x_1(t) = B_1(t)x(\mathrm{at})$$

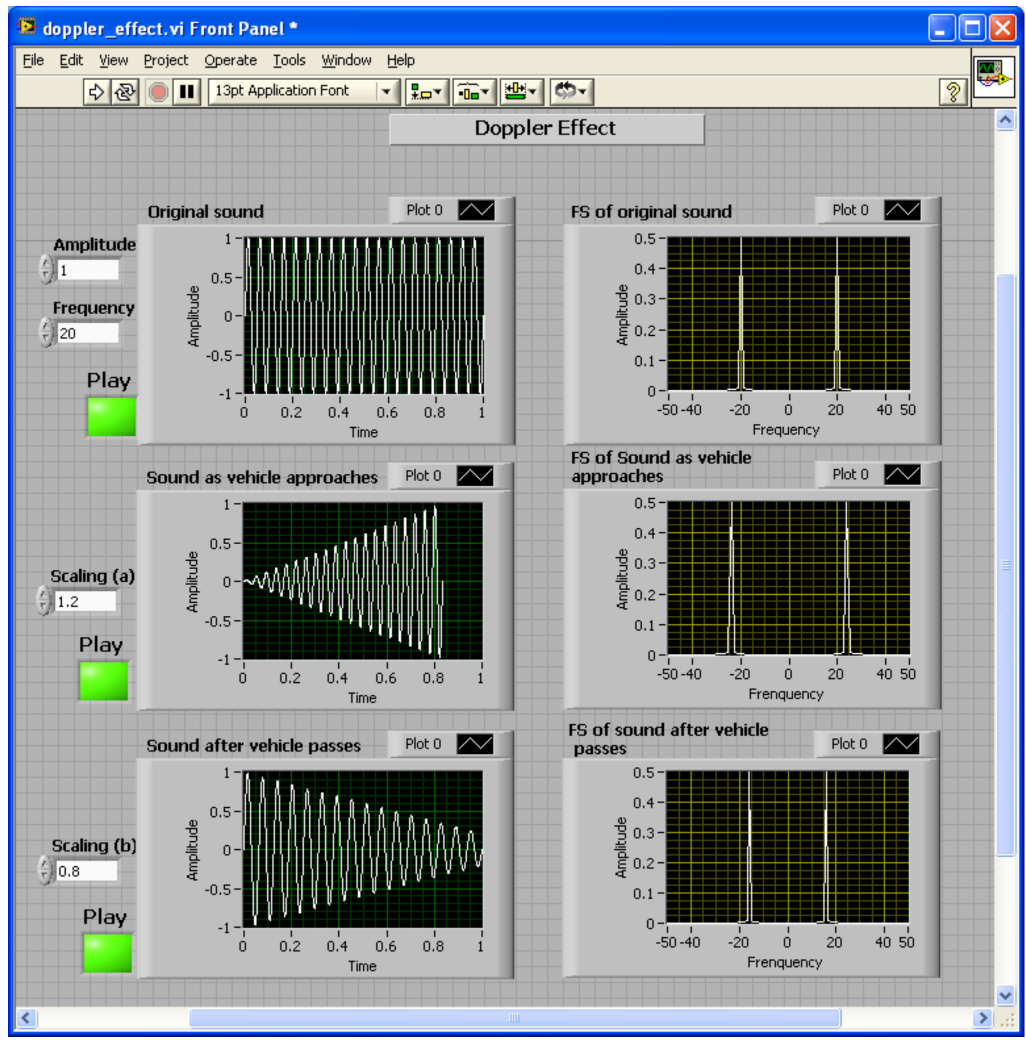
where $B_1(t)$is an increasing function of time (assuming a linear increment with time) and $a$is the scaling factor having a value greater than 1. When the vehicle passes by, one can describe the signal by

Equation:

$$x_2(t) = B_2(t)x(\mathrm{bt})$$

where $B_2(t)$is a decreasing function of time (assuming a linear decrement with time) and $b$is the scaling factor having a value less than 1.

First, generate a signal and create an upscale and a downscale version of it. Observe the Fourier series for all the signals. Set the amplitude and frequency of the original signal and the scaling factors as controls. In addition, play the sounds using the LabVIEW **Play Waveform** function. [link] shows a possible front panel for this type of system.



Front Panel of a Doppler Effect System

Solution:

Insert Solution Text Here

Exercise:

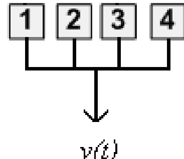
Problem: Synthesis of Electronic Music

In electronic music instruments, sound generation is implemented via synthesis. Different types of synthesis techniques such as additive synthesis, subtractive synthesis and frequency modulation (FM) synthesis are used to create audio waveforms. The simplest type of synthesis is additive synthesis, where a composite waveform is created by summing sine wave components, which is basically the

inverse Fourier series operation. However, in practice, to create a music sound with rich harmonics requires adding a large number of sine waves, which makes the approach inefficient computationally. To avoid adding a large number of sine waves, modulation with addition is used. This exercise involves the design of algorithms used in the Yamaha DX7 music synthesizer, which debuted in 1983 as the first commercially available digital synthesizer.

The primary functional circuit in DX7 consists of a digital sine wave oscillator plus a digital envelope generator. Let us use additive synthesis and frequency modulation to achieve synthesis with six configurable operators. When one adds together the output of some operators, an additive synthesis occurs, and when one connects the output of one operator to the input of another operator, a modulation occurs.

In terms of block diagrams, the additive synthesis of a waveform with four operators is illustrated in [link].



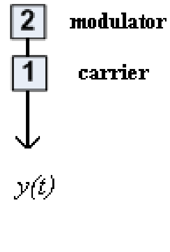
Additive Synthesis

The output for the combination shown in [link] can be written as

Equation:

$$y(t) = A_1 \sin(\omega_1 t) + A_2 \sin(\omega_2 t) + A_3 \sin(\omega_3 t) + A_4 \sin(\omega_4 t)$$

[link] shows the FM synthesis of a waveform with two operators.



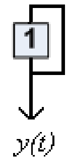
FM Synthesis

The output for the combination shown in this figure can be written as

Equation:

$$y(t) = A_1 \sin(\omega_1 t + A_2 \sin(\omega_2 t))$$

Other than addition and frequency modulation, one can use feedback or self-modulation in DX7, which involves wrapping back and using the output of an operator to modulate the input of the same operator as shown in [link].



Self-Modulation

The corresponding equation is

Equation:

$$y(t) = A_1 \sin(\omega_1 t + y(t))$$

Different arrangements of operators create different algorithms. [link] displays the diagram of an algorithm.

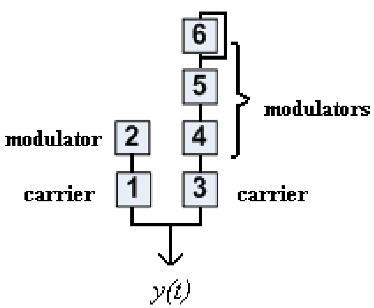


Diagram of an Algorithm

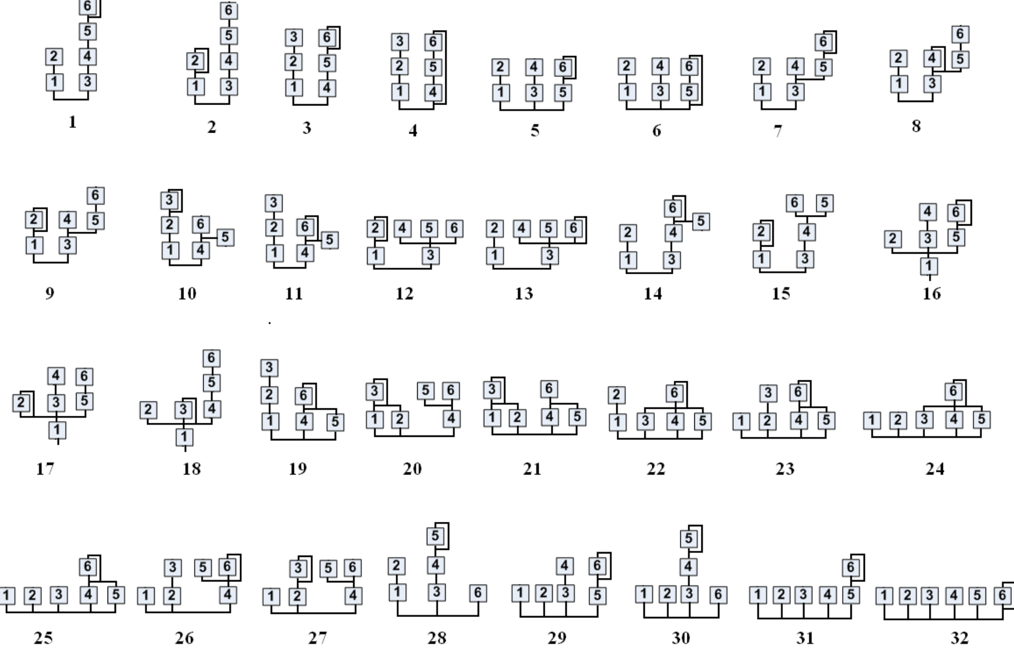
And the output for this algorithm can be written as

Equation:

$$y(t) = A_1 \sin(\omega_1 t + A_2 \sin(\omega_2 t)) + A_3 \sin(\omega_3 t + A_4 \sin(\omega_4 t + A_5 \sin(\omega_5 t + y_6(t))))$$

With DX7, one can choose from 32 different algorithms. As one moves from algorithm No. 32 to algorithm No. 1, the harmonics complexity increases. In algorithm No. 32, all six operators are combined using additive synthesis with a self modulator generating the smallest number of harmonics. [link] shows the diagram for all 32 combinations of operators. More details on music synthesis and the Yamaha DX7 synthesizer can be found in the [link]-[link].

Next, explore designing a system with six operators and set their amplitude and frequency as controls. By combining these operators, construct any three algorithms, one from the lower side (for example, algorithm No. 3), one from the middle side (for example, algorithm No. 17) and the final one from the upper side (for example, algorithm No. 30). Observe the output waves in the time and frequency domains (find the corresponding Fourier series).



32 Algorithms in the Yamaha DX7

Solution:

Insert Solution Text Here

Continuous-Time Fourier Transform

In this lab, we learn how to compute the continuous-time Fourier transform (CTFT), normally referred to as Fourier transform, numerically and examine its properties. Also, we explore noise cancellation and amplitude modulation as applications of Fourier transform.

Properties of CTFT

The continuous-time Fourier transform (CTFT) (commonly known as Fourier transform) of an aperiodic signal $x(t)$ is given by
Equation:

$$X(\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} \mathrm{dt}$$

The signal $x(t)$ can be recovered from $X(\omega)$ via this inverse transform equation
Equation:

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\omega) e^{j\omega t} d\omega$$

Some of the properties of CTFT are listed in [link].

Properties	Time domain	Frequency domain
Time shift	$x(t - t_0)$	$X(\omega)e^{-j\omega t_0}$

Time scaling	$x(\text{at})$	$\frac{1}{\lvert a \rvert} X\left(\frac{\omega}{a}\right)$
Linearity	$a_1 x_1(t) + a_2 x_2(t)$	$a_1 X_1(\omega) + a_2 X_2(\omega)$
Time convolution	$x(t) * h(t)$	$X(\omega) H(\omega)$
Frequency convolution	$x(t) h(t)$	$X(\omega) * H(\omega)$

Properties of CTFT

Refer to signals and systems textbooks [link] - [link] for more theoretical details on this transform.

Numerical Approximations to CTFT

Assuming that the signal $x(t)$ is zero for $t<0$ and $t \geq T$, we can approximate the CTFT integration in Equation (1) as follows:
Equation:

$$\int_{-\infty}^{\infty} x(t) e^{-j\omega t} \text{dt} = \int_{0}^{T} x(t) e^{-j\omega t} \text{dt} \approx \sum_{n=0}^{N-1} x(n\tau) e^{-j\omega n\tau} \tau$$

where $T = N\tau$ and N is an integer. For sufficiently small τ, the above summation provides a close approximation to the CTFT integral. The summation $\sum_{n=0}^{N-1} x(n\tau) e^{-j\omega n\tau}$is widely used in digital signal processing (DSP), and both LabVIEW MathScript and LabVIEW have a built-in function for it called `fft`. In a .m file, if N samples $x(n\tau)$ are stored in a vector x, then the function call

```
>>xw=tau*fft(x)
```

calculates
Equation:

$$X_\omega(k+1) = \tau \sum_{n=0}^{N-1} x(n\tau) e^{-j\omega_k n\tau} \quad 0 \le k \le N-1$$
$$\approx X(\omega_k)$$

where
Equation:

$$\omega_k = \begin{cases} \frac{2\pi k}{N\tau} & 0 \le k \le \frac{N}{2} \\ \frac{2\pi k}{N\tau} - \frac{2\pi}{\tau} & \frac{N}{2}+1 \le k \le N-1 \end{cases}$$

with N assumed to be even. The `fft` function returns the positive frequency samples before the negative frequency samples. To place the frequency samples in the right order, use the function `fftshift` as indicated below:

```
>>xw=fftshift(tau*fft(x))
```

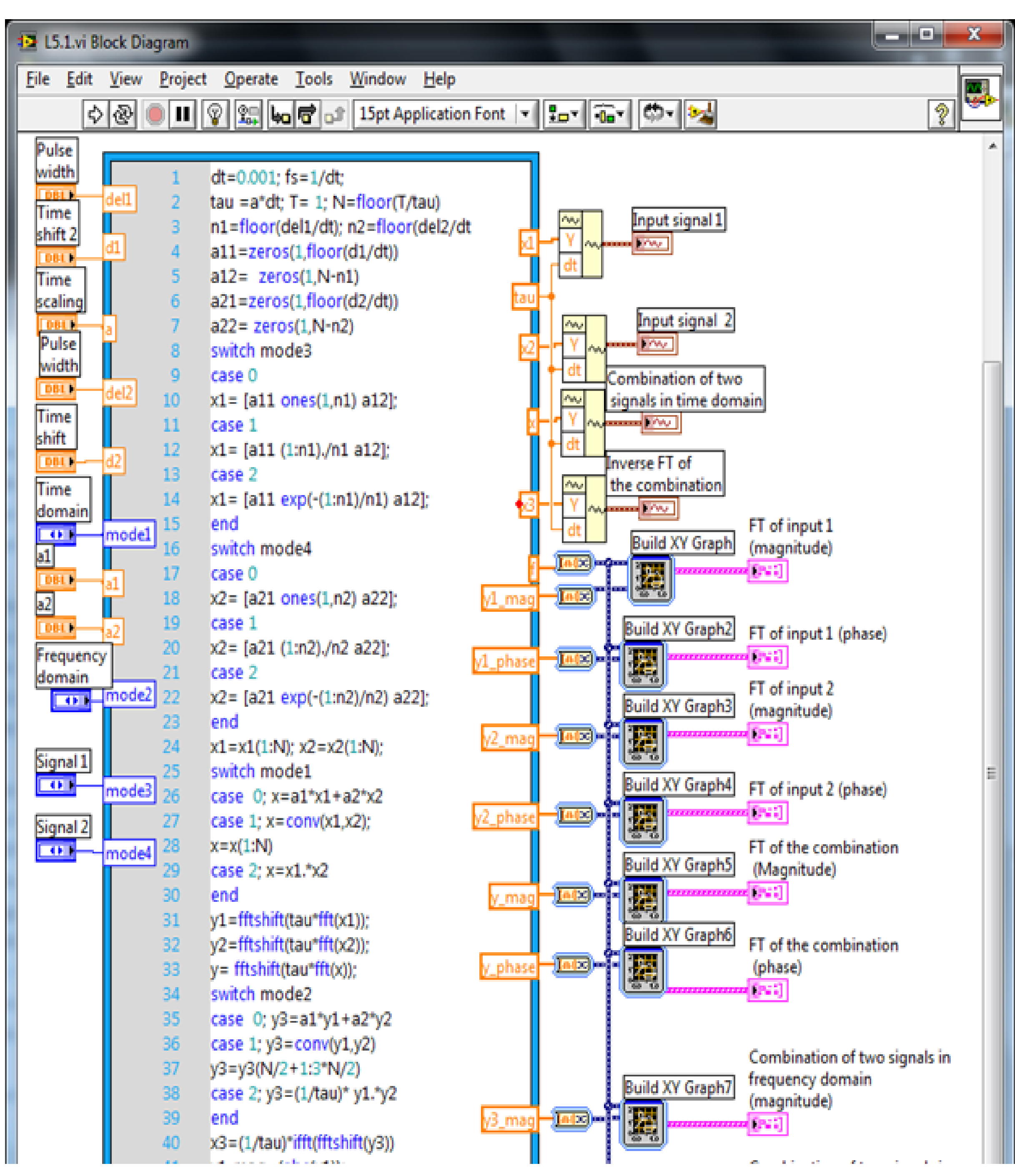
Note that $X(\omega)$ is a vector (actually, a complex vector) of dimension N. $X(\omega)$ is complex in general despite the fact that $x(t)$ is real-valued. The magnitude of $X(\omega)$ can be computed using the function `abs` and the phase of $X(\omega)$ using the function `angle`.

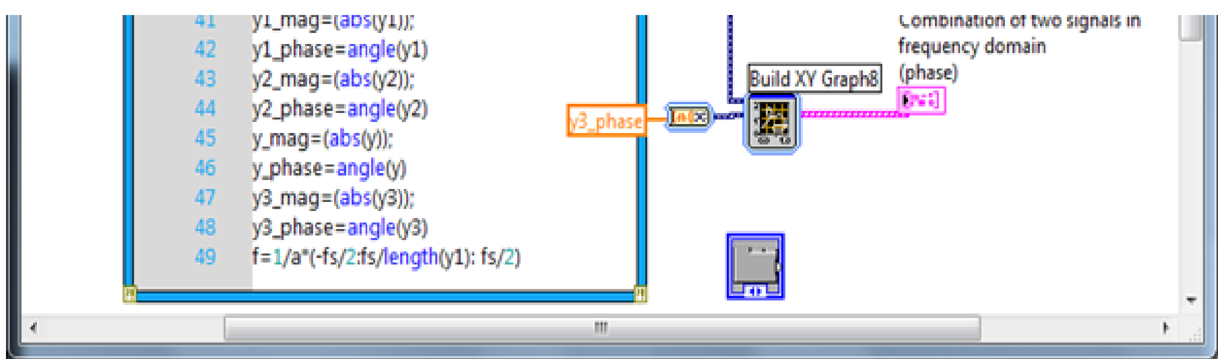
Properties of CTFT

The example covered in this section provides an implementation of CTFT and its properties. As mentioned earlier, programming environments can generate and work with only discrete values arranged in arrays. Thus, to get a continuous-time representation of a signal, use a very small value of time increment dt. For example, dt=0.001 means there are 1000 discrete samples in 1 second, which provides a good approximation to represent a low-frequency signal. However, when working with very high-frequency signals, one should decrease the value of dt further to ensure there are enough samples to represent the signal in a continuous fashion over a specified duration.

[link] shows the example of the completed block diagram for the CTFT (or FT) and its properties. This particular VI is capable of finding the FT of a rectangular and a triangular pulse. Create two input signals using the LabVIEW MathScript functions ones and zeros, which are combined in the time domain. Use a case structure to select the combination method (linear combination, convolution or multiplication) and the parameter mode1 to serve as an input that is connected to an **Enum Control(Controls → Modern→Ring & Enum →Enum)**. Use parameters mode3 and mode4, which are connected to two Enum controls, to select the input signal type. Also set Pulse width, Time shift and Time scale as control parameters. Pulse width controls the number of ones in the pulse, which is used to increase or decrease the pulse width. Time shift adds zeros before the pulse to provide a time delay. Time scale is set to be multiplied with the time increment (dt) to ensure appropriate scaling of the pulse. Use the LabVIEW MathScript function `fft` to determine the FT of the continuous signal. Combine the signals in the frequency domain and control the combination method (linear combination, convolution or multiplication) via the parameter mode2. Compute the FT of the time domain combinations and the inverse FT of the frequency domain combinations using the functions `fft` and `ifft`. To shift the zero-frequency component to the center of the spectrum, use the LabVIEW MathScript function `fftshift`. Finally, determine the magnitude and phase of the FT using the functions `abs` and `angle`,

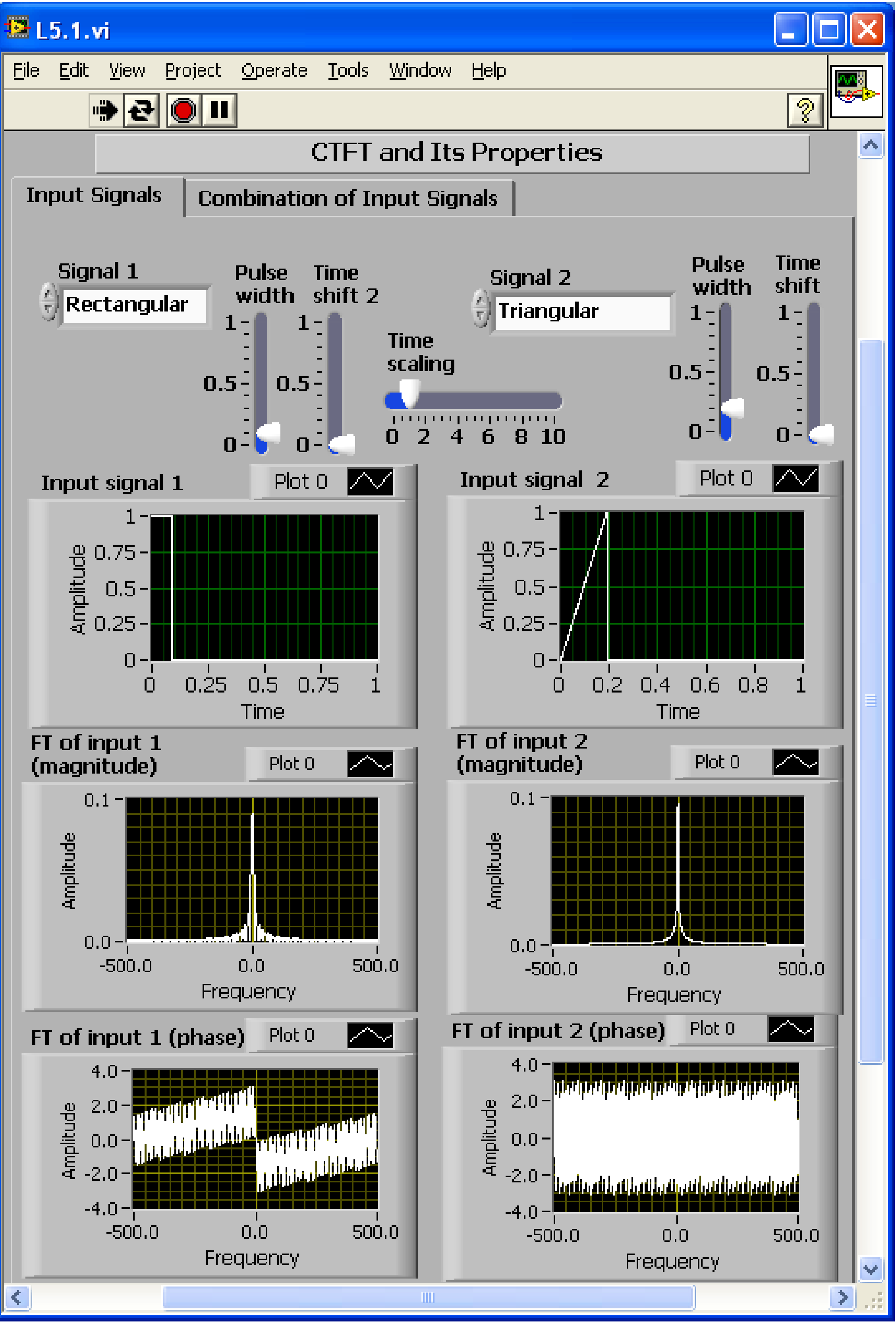
respectively. Display the input signals and their combinations using a **Build Waveform** function **(Functions → Programming →Waveforms →Build Waveforms)** and a **Waveform Graph(Controls →Modern →Graph →Waveform Graph)**. Also, display the spectrum magnitude and phase using a waveform graph.



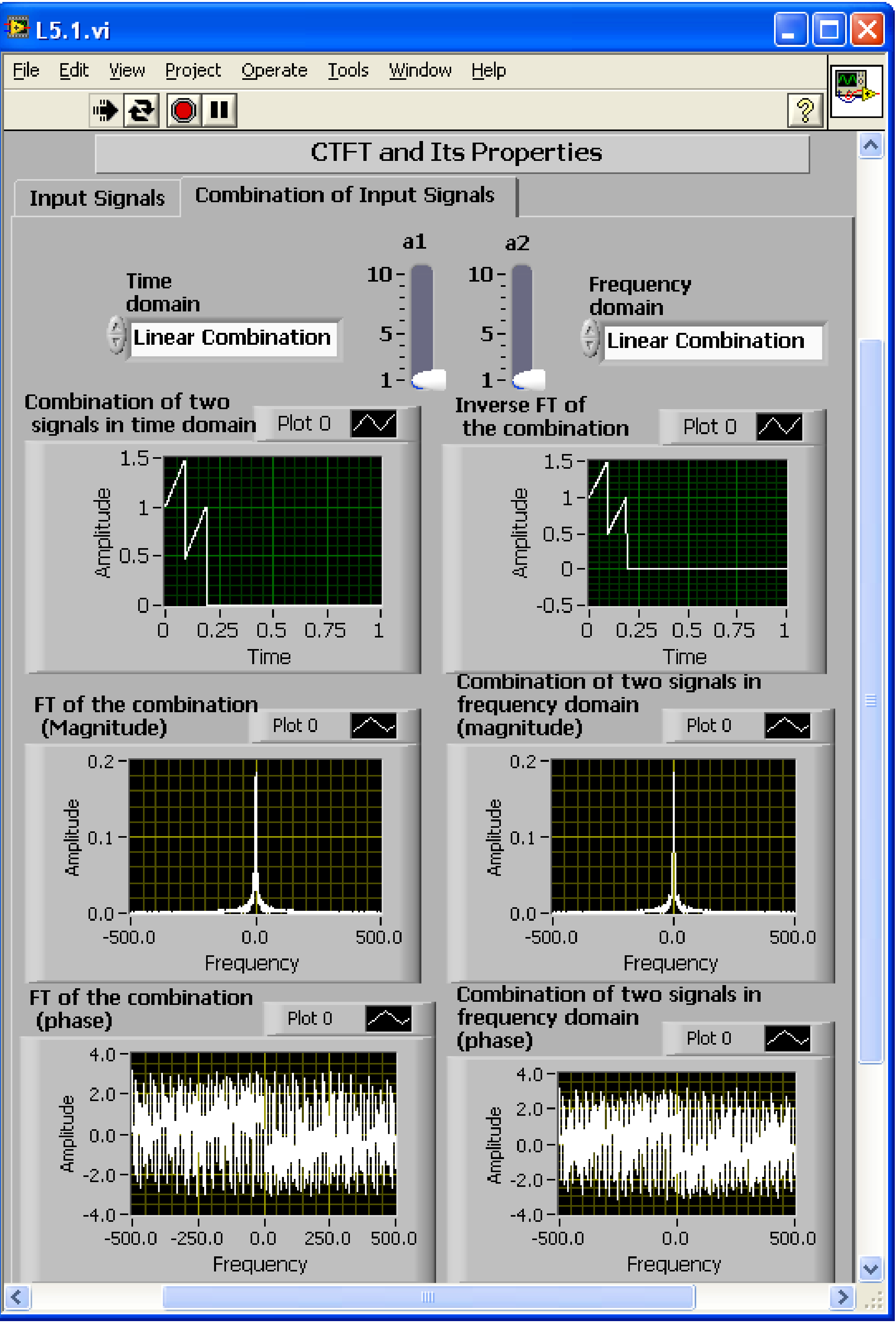


Block Diagram of CTFT and Its Properties

[link] and [link] shows the front panel of the above system. Use controls named Pulse width, Time shift and Time scaling to change the waveforms in the time domain. Three waveform graphs for Input signal, Magnitude of FT and Phase of FT also appear in the front panel shown. With the specified front panel controls, one can easily verify CTFT properties. To begin with, run the program in continuous mode using the **Run Continuously** button.


L5.1.vi
File Edit View Project Operate Tools Window Help
CTFT and Its Properties
Input Signals
Combination of Input Signals
Signal 1
Rectangular
Pulse width
Time shift 2
Time scaling
Signal 2
Triangular
Pulse width
Time shift
Input signal 1
Input signal 2
Plot 0
Amplitude
Time
FT of input 1 (magnitude)
FT of input 2 (magnitude)
Frequency
FT of input 1 (phase)
FT of input 2 (phase)

Front Panel of CTFT and Its Properties: Input Signals Tab

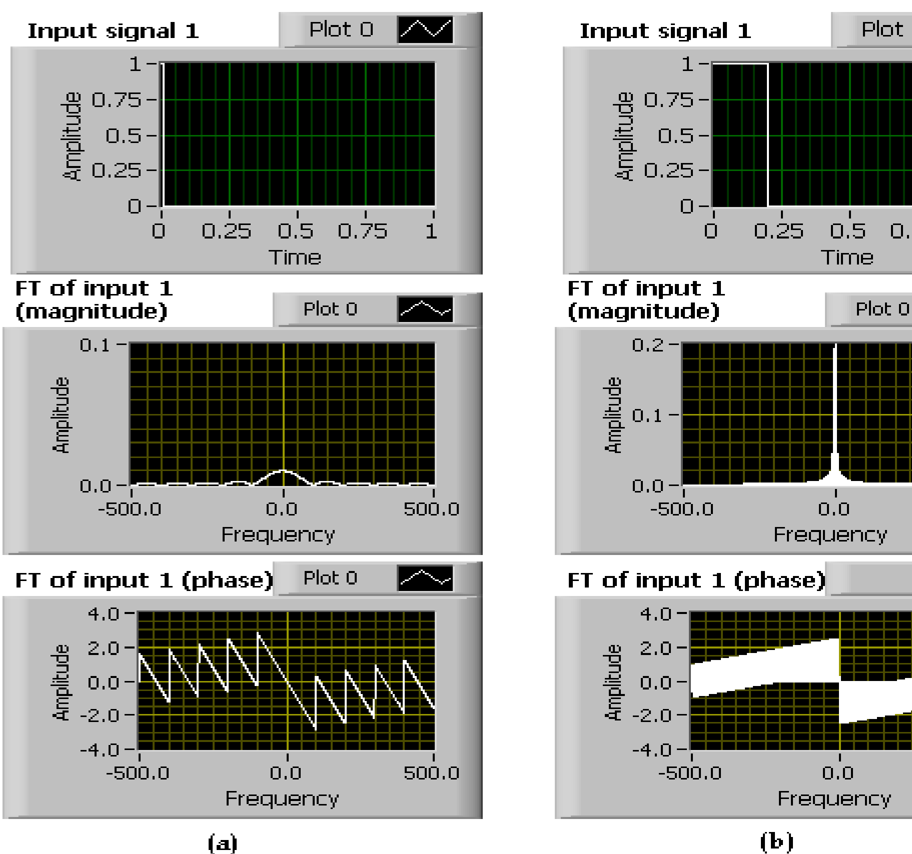


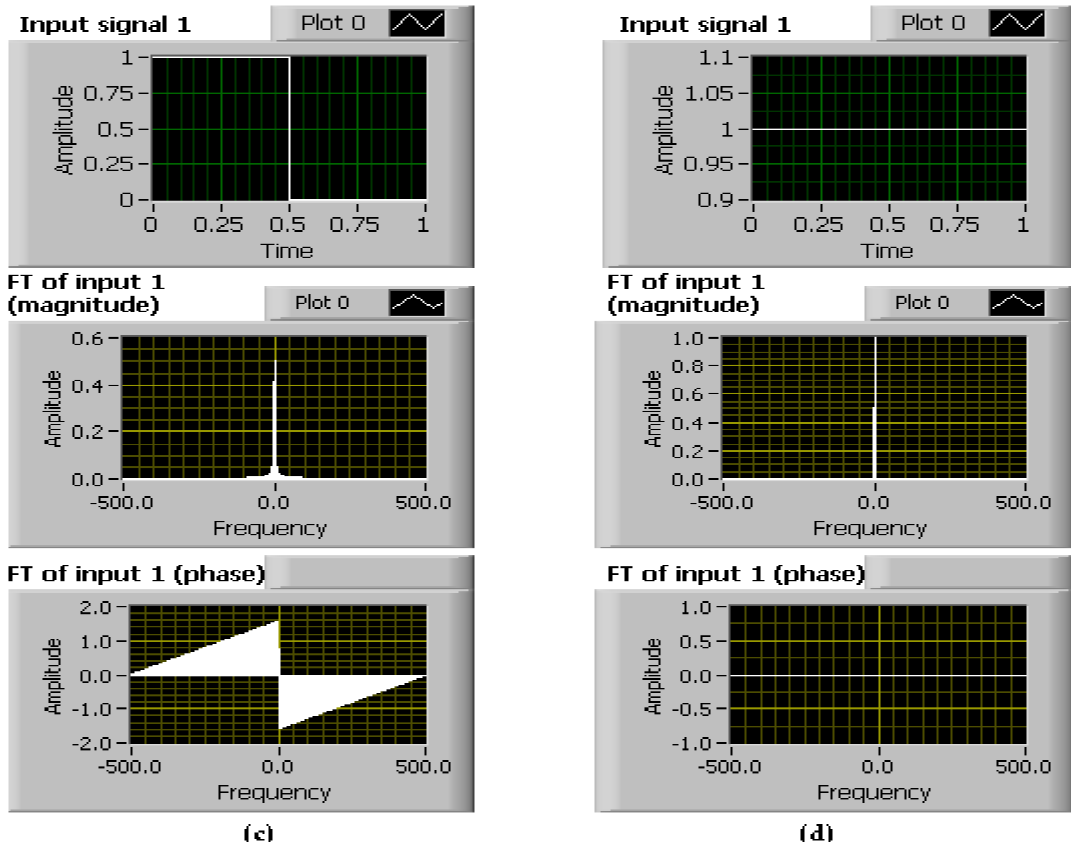
L5.1.vi
File Edit View Project Operate Tools Window Help
CTFT and Its Properties
Input Signals
Combination of Input Signals
a1
a2
Time domain
Linear Combination
Frequency domain
Linear Combination
Combination of two signals in time domain
Inverse FT of the combination
FT of the combination (Magnitude)
Combination of two signals in frequency domain (magnitude)
FT of the combination (phase)
Combination of two signals in frequency domain (phase)
Plot 0
Amplitude
Time
Frequency

Front Panel of CTFT and Its Properties: Combination of Input Signals Tab

Varying Pulse Width

Keep the default values of Time shift (=0) and Time scaling (=1) and vary the Pulse width of the rectangular pulse. First, set the value of the Pulse width to its minimum value (=0.01) and then increase it. Observe that increasing the Pulse width in the time domain decrements the width in the frequency domain (see [link]). When the Pulse width is set to its maximum value (=1) in the frequency domain, only one value can be seen at the center frequency indicating the signal is of DC type (refer to Properties of CTFT section of Chapter 5).

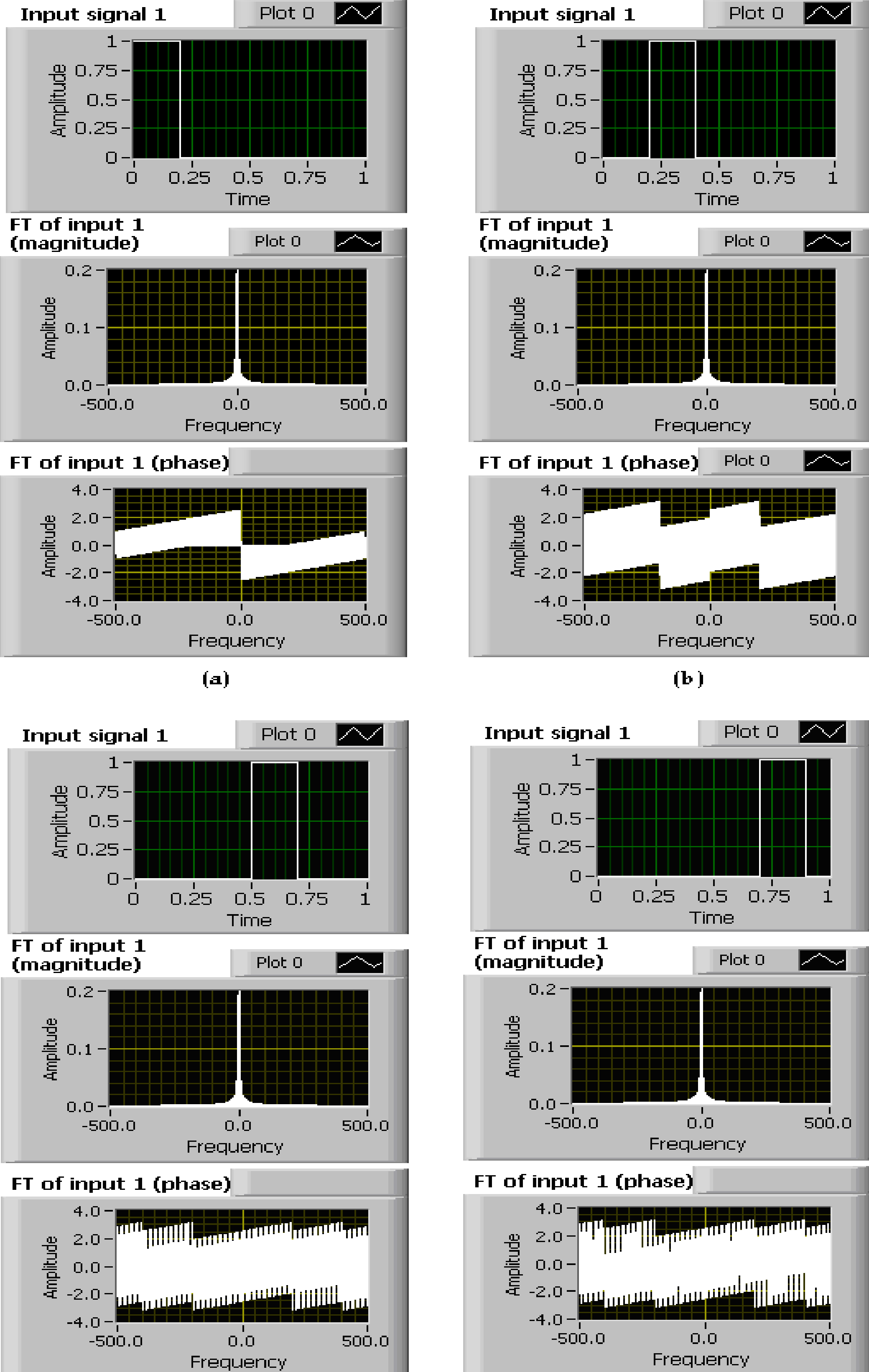




Magnitude Spectrum for Different Pulse Widths: (a) 0.01, (b) 0.2, (c) 0.5, (d) 1

Time Shift

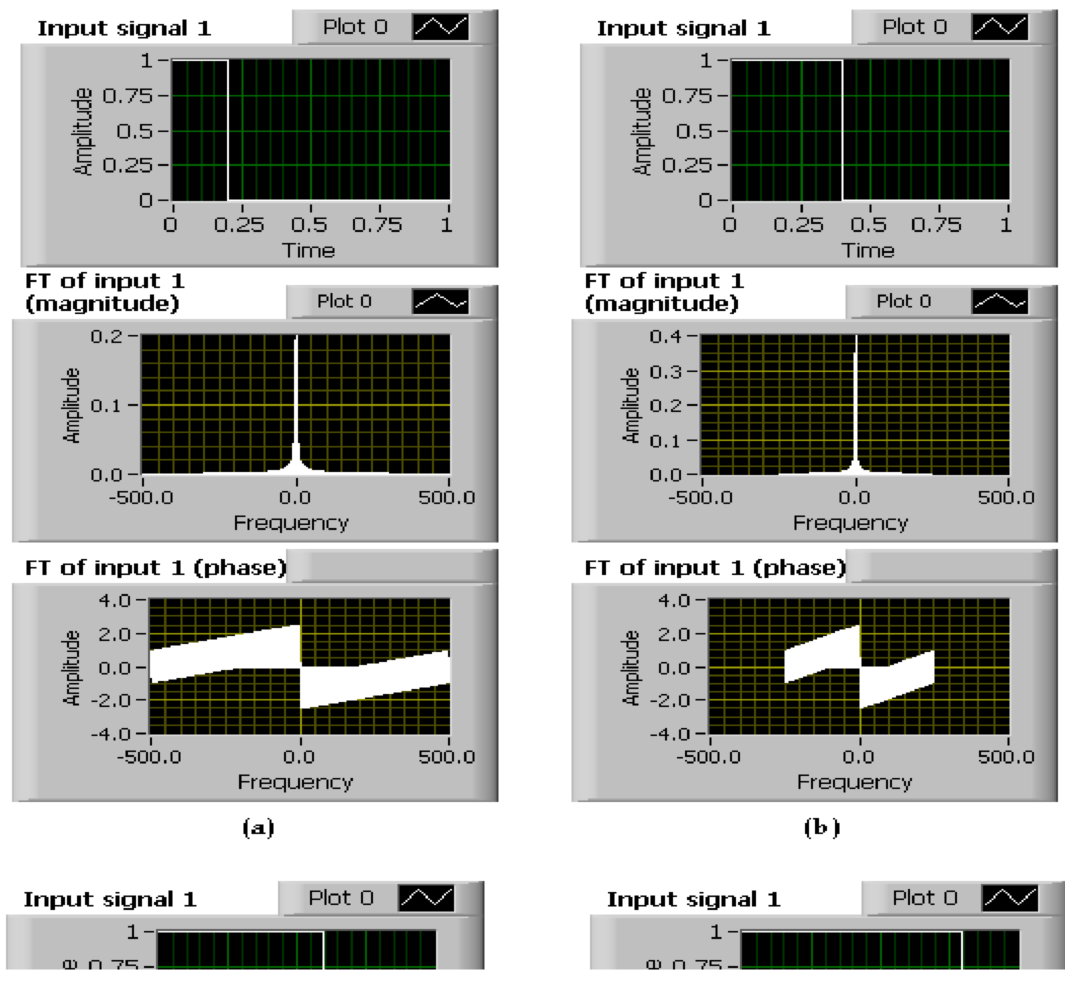
Next, for a fixed pulse width, vary the time shift. Observe that the phase spectrum changes but the magnitude spectrum remains the same. If the signal $x(t)$ is shifted by a constant t_0 , its FT magnitude does not change, but the term $-\omega t_0$ gets added to its phase angle. This verifies the time-shifting property of FT as stated in Properties of CTFT section of Chapter 5 (see [link]).

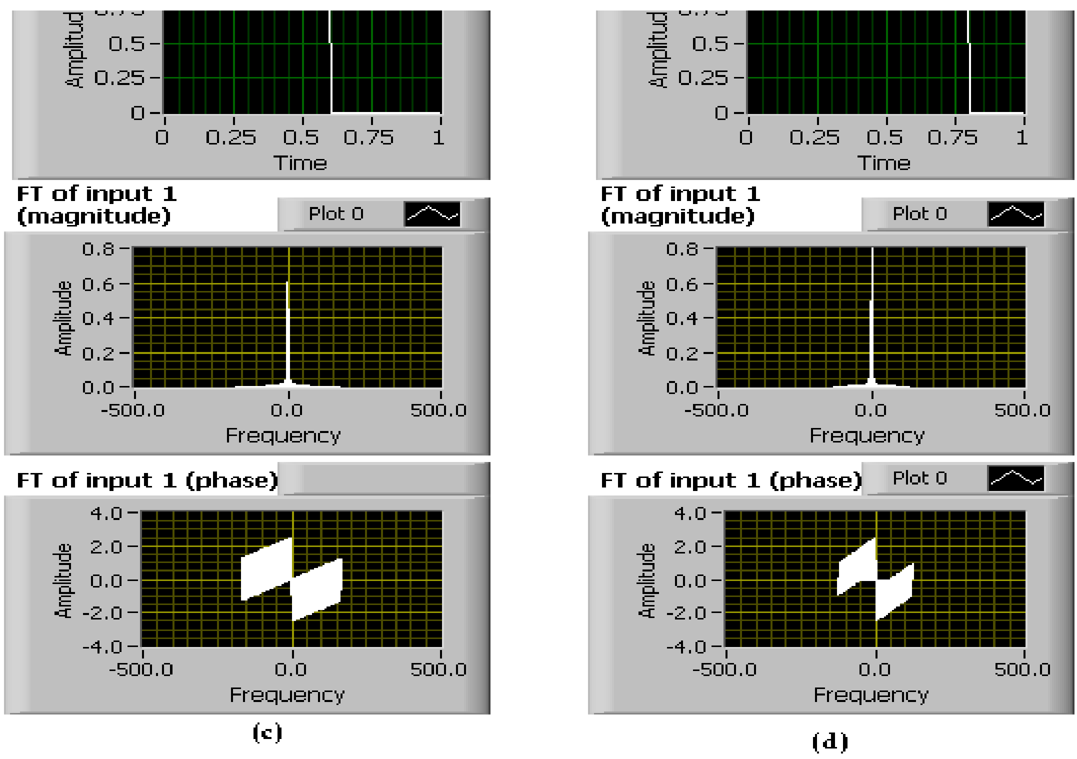

Input signal 1
Plot 0
Amplitude
Time
FT of input 1 (magnitude)
Frequency
FT of input 1 (phase)
(a)
(b)
(c)
(d)

Magnitude and Phase Spectrum for Different Time Shifts: (a) 0, (b) 0.2, (c) 0.5, (d) 0.7

Time Scaling

Observe that increasing the control Time scaling makes the spectrum wider. This indicates that compressing the signal in the time domain leads to expansion in the frequency domain. This verifies the time-scaling property of FT as stated in Properties of CTFT section of Chapter 5 (see [link]).

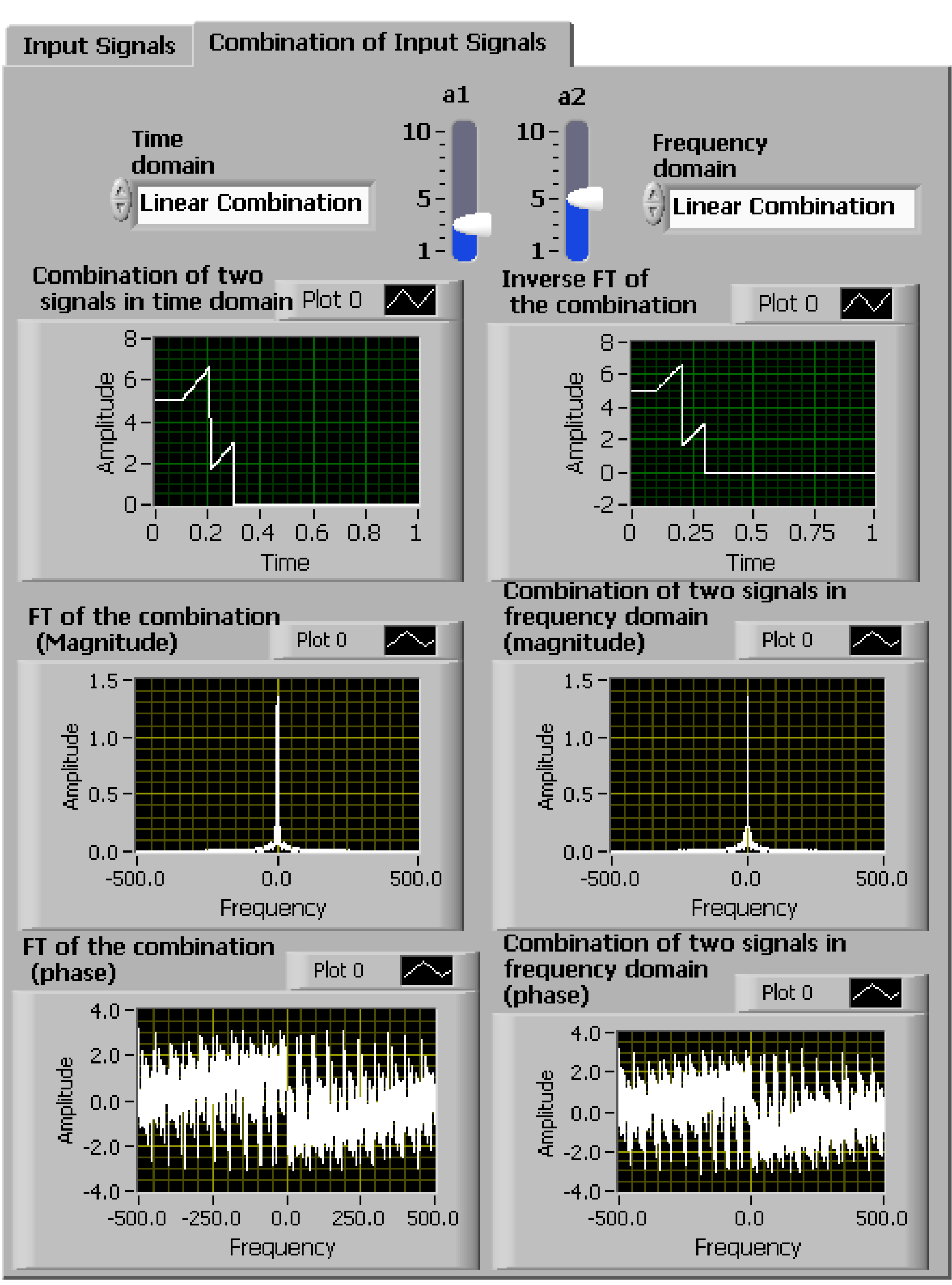




Magnitude Spectrum for Different Time Scalings: (a) 1, (b) 2, (c) 3, (d) 4

Linearity

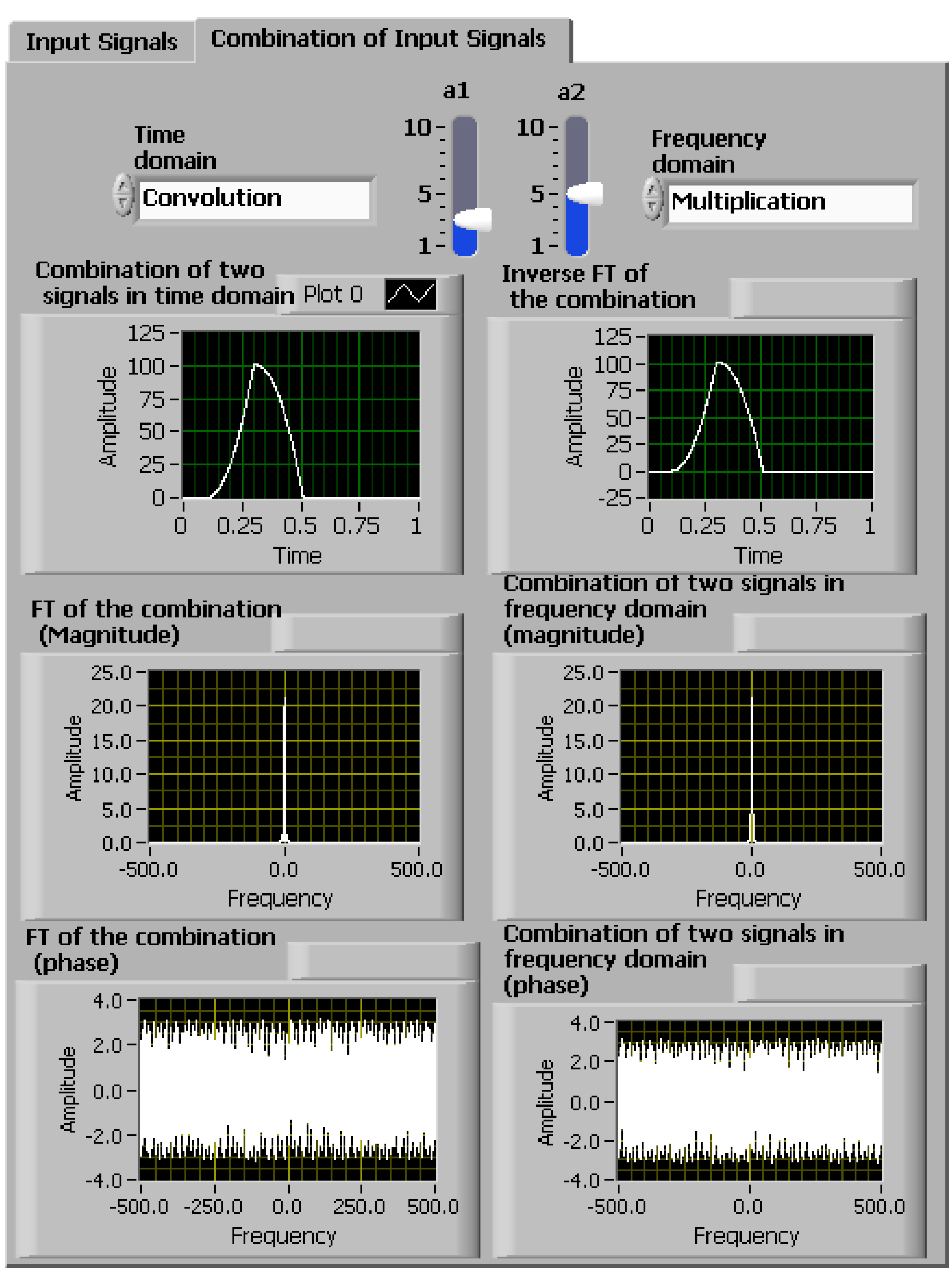
Here, combine two signals to examine the linearity property of FT. Select Linear Combination for the Time domain and Frequency domain combination method. This selection combines two time signals, $x_1(t)$and $x_2(t)$, linearly with the scaling factors, $a_1$and a_2, producing a new signal, $a_1x_1(t)+a_2x_2(t)$. [link] displays the FT of this linear combination. The linear combination in the frequency domain produces a new signal, $a_1X_1(\omega)+a_2X_2(\omega)$. [link] also displays the inverse FT of this combination. Observe that both combinations produce the same result in the time and frequency domains, as indicated by the linearity property stated in Properties of CTFT section of Chapter 5.



Verifying the Linearity Property of CTFT

Time Convolution

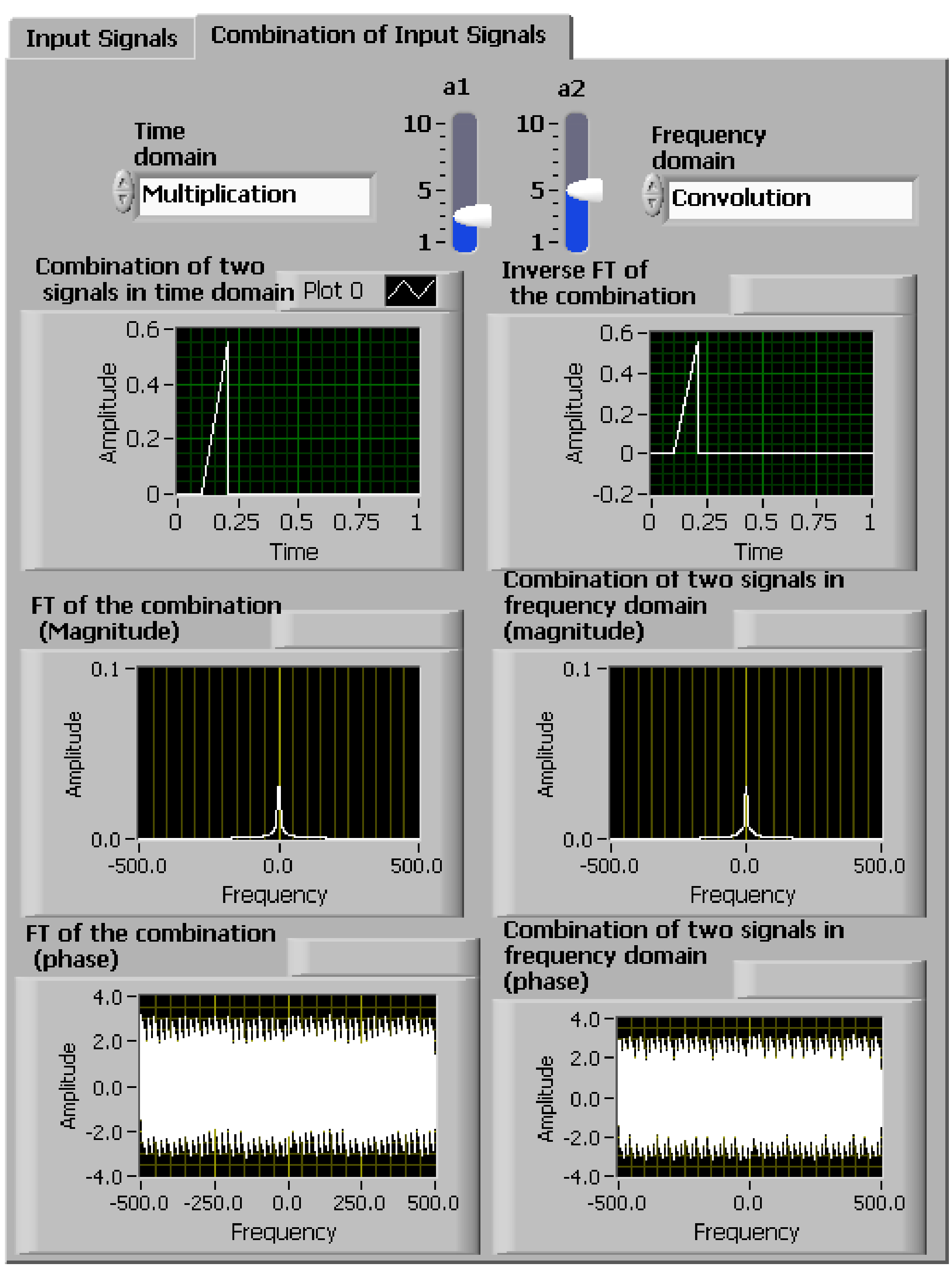
In this part, convolve two signals in the time domain to examine the time-convolution property of FT. Select Convolution for Time domain and Multiplication for Frequency domain. This selection produces and displays a new signal, $x_1(t) * x_2(t)$, by convolving the two time signals $x_1(t)$and $x_2(t)$. Multiplication in the frequency domain produces a new signal, $X_1(\omega)X_2(\omega)$. The inverse FT of this multiplied signal is also displayed on the right. Note that both combinations produce the same outcome in the time and frequency domains. This verifies the time-convolution property stated in the Properties of CTFT section of Chapter 5 (see [link]).



Verifying the Time-Convolution Property of CTFT

Frequency Convolution

Convolve two signals in the frequency domain to examine the frequency-convolution property of FT. Select Convolution for Frequency domain and Multiplication for Time domain. This selection convolves the two time signals $X_1(\omega)$and $X_2(\omega)$ to produce a new signal, $X_1(\omega) * X_2(\omega)$. The inverse FT of the convolved signal is displayed. Multiplication in Time domain produces a new signal, $x_1(t)x_2(t)$. The FT of this multiplied signal is also displayed. Note that both combinations produce the same outcome in the time and frequency domains. This verifies the frequency-convolution property stated in the Properties of CTFT section of Chapter 5 (see [link]).



Verifying the Frequency-Convolution Property of CTFT

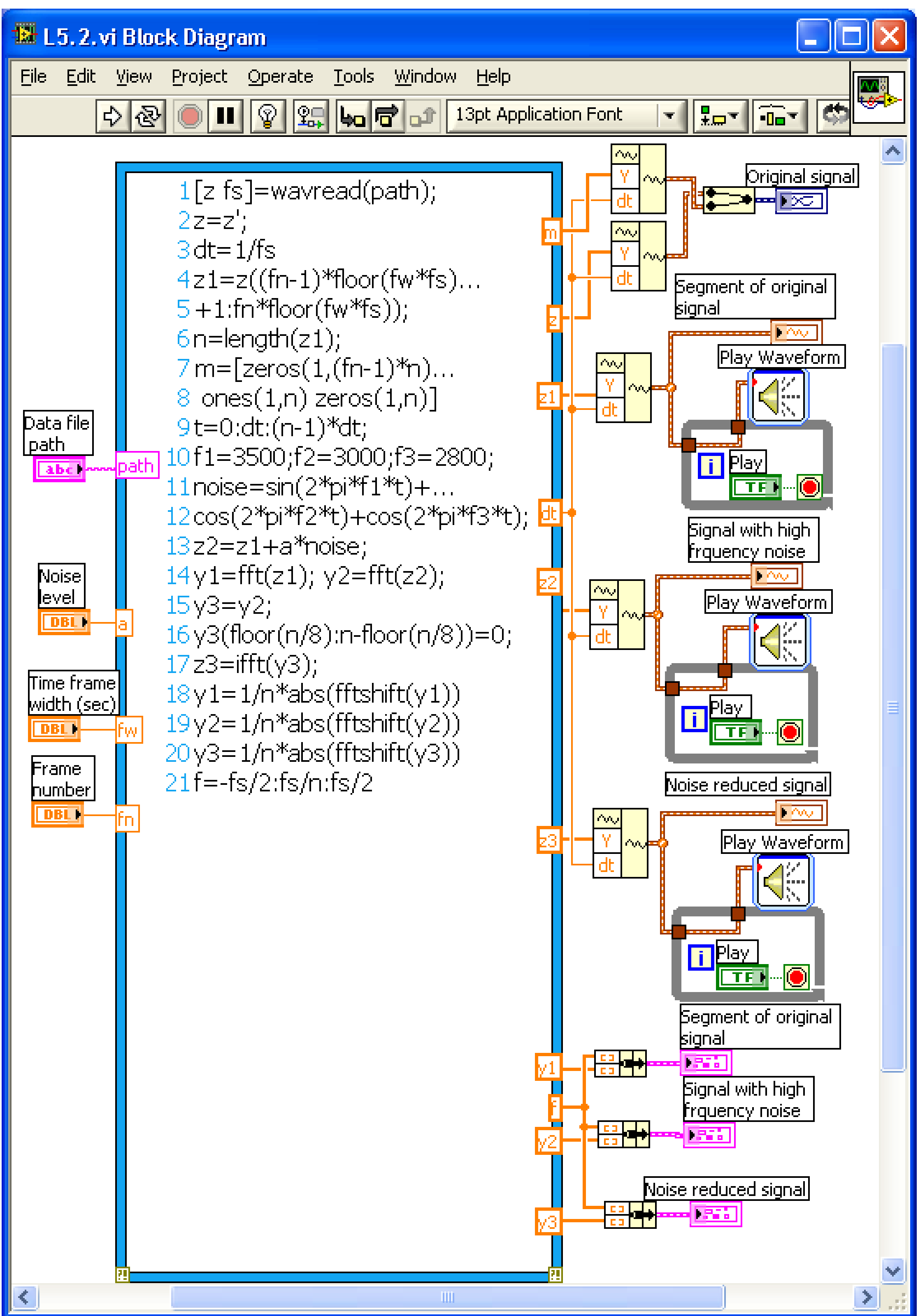
Noise Reduction

When a signal passes through a channel, it normally gets corrupted by channel noise. Various electronic components used in a transmitter or receiver may also cause additional noise. Noise reduction is an important aspect of any signal processing system. Lab 7 features noise reduction techniques using digital and analog filters. This section presents a simple technique to reduce high-frequency noise.

[link] shows the completed block diagram of a noise reduction system. Consider a speech signal sampled at 8 kHz. Add some high-frequency noise to this signal and then remove the high-frequency components in the frequency domain. Finally, move the signal back into the time domain using the inverse FT. Use the LabVIEW MathScript function wavread to read a wave file specified by the string Path and return the sampled data at a specified sampling rate. A **String Control(Controls → Modern →String & Path →String Control)** can be added to the input Path to provide the path name for the speech data file. Use two more controls named Time frame width and Frame number to extract a segment of the speech signal before computing Fourier transform. Add together three sine and cosine waves with frequencies of 3.5, 3 and 2.8 kHz to create a high-frequency noise. Then add a scaled version of the noise signal to the signal with the Scaling parameter set as a control. Compute the FT of the Noise added signal using the function `fft`.

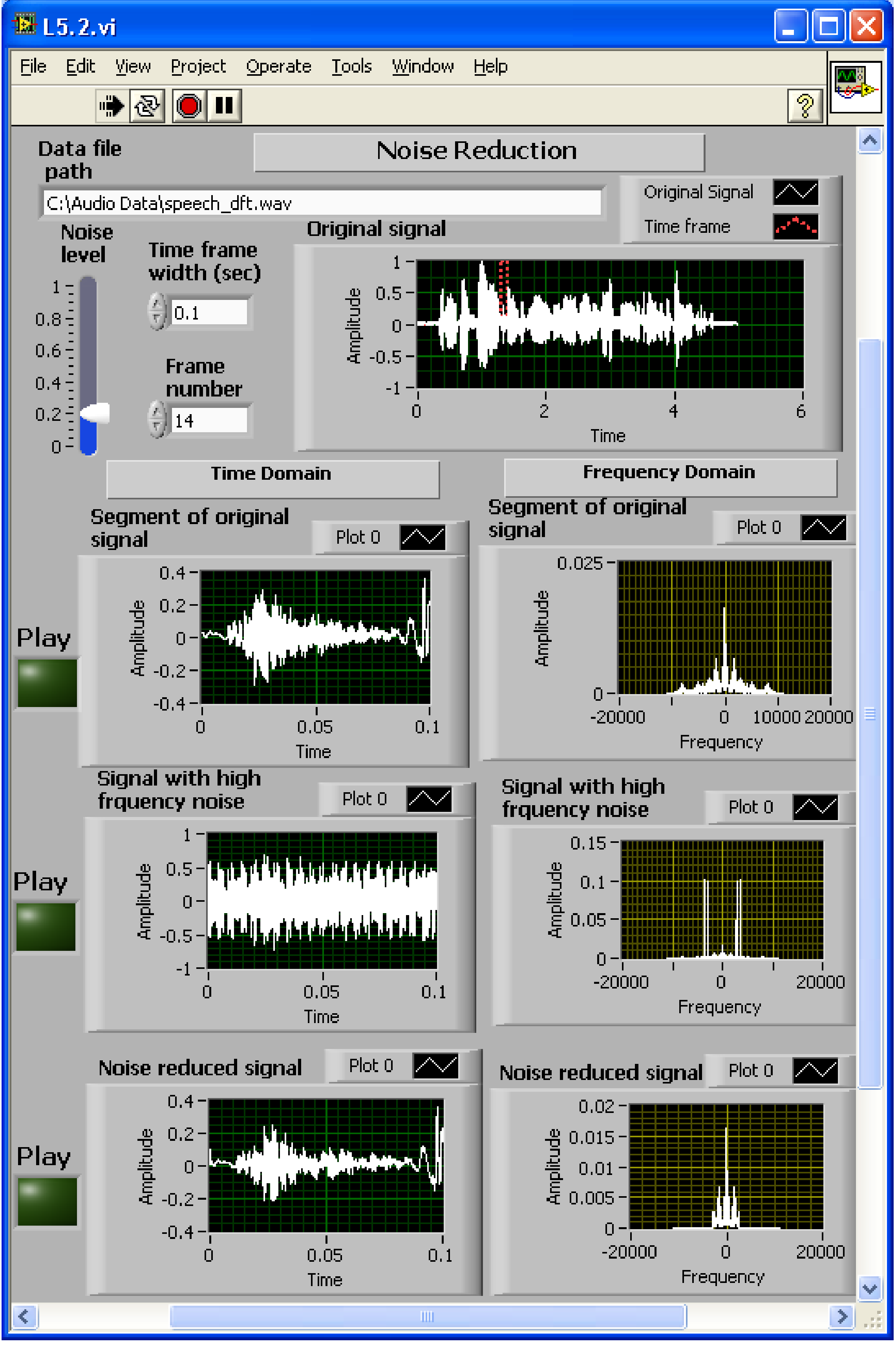
To remove the high-frequency noise components, use a simple lowpass filter by removing the frequency components over a certain threshold (50 percent, for example). After removing the high-frequency components, transform the signal back into the time domain using the function `ifft`. To get a display of the absolute and centered frequency spectrum, use the functions abs and `fftshift`. The signals are displayed in the time domain using the functions **Build Waveform** and **Waveform Graph**. To be able to hear the speech signals, use the function **Play Waveform(Functions →Programming →Graphics & Sound →Sound→Output →Play Waveform)**. Connect the time domain signals to this function via the while loop structure. Connect a **Boolean control(Controls →Modern →Boolean →Push Button)** to the loop control, which acts as a play switch for the sound signal. The signals are also displayed in the frequency domain using the functions **Bundle(Functions →Programming →Cluster, Class &**

Variant → Bundle) and **XY Graph(Controls →Modern →Graphs →XY Graph)**.



Block Diagram of a Noise Reduction System

[link] shows the front panel of the system. Inside the Data File Path control, the location of the speech data file is specified. Three graphs for the Original signal, Noise added signal and Noise reduced signal are shown in both the time and frequency domains. Use the noise level control to allow setting the amount of noise added to the original signal. After running the program, click on the **Play** button next to each signal. Hear the Original Signal and the Noise added signal. Notice that an unpleasant high-pitched noise gets added to the signal. If the noise level is set more than 0.5, the Original Signal becomes very difficult to hear. Next, hear the Noise reduced signal, which is similar to the Original Signal. The Noise reduced signal is not exactly the same as the Original Signal because some high-frequency components are also removed along with the noise.



L5.2.vi
File Edit View Project Operate Tools Window Help
Data file path
C:\Audio Data\speech_dft.wav
Noise Reduction
Original Signal
Time frame
Original signal
Noise level
Time frame width (sec)
0.1
Frame number
14
Amplitude
Time
Time Domain
Frequency Domain
Segment of original signal
Plot 0
Frequency
Play
Signal with high frquency noise
Noise reduced signal

Front Panel of a Noise Reduction System

Amplitude Modulation

In this section, we examine amplitude modulation and demodulation applications. For transmission purposes, signals are often modulated with a high-frequency carrier. A typical amplitude modulated signal can be described by

Equation:

$$x(t) = x_m(t)\cos(2\pi f_c t)$$

where $x_m(t)$is called the message waveform, which contains the data of interest, and $f_c$is the carrier wave frequency. Using the fact that

Equation:

$$\cos(2\pi f_c t) = \frac{1}{2}(e^{2\pi f_c t} + e^{-2\pi f_c t}) = \frac{1}{2}(e^{\omega_c t} + e^{-\omega_c t})$$

and the frequency shift property of CTFT, one can easily derive the CTFT of to be

Equation:

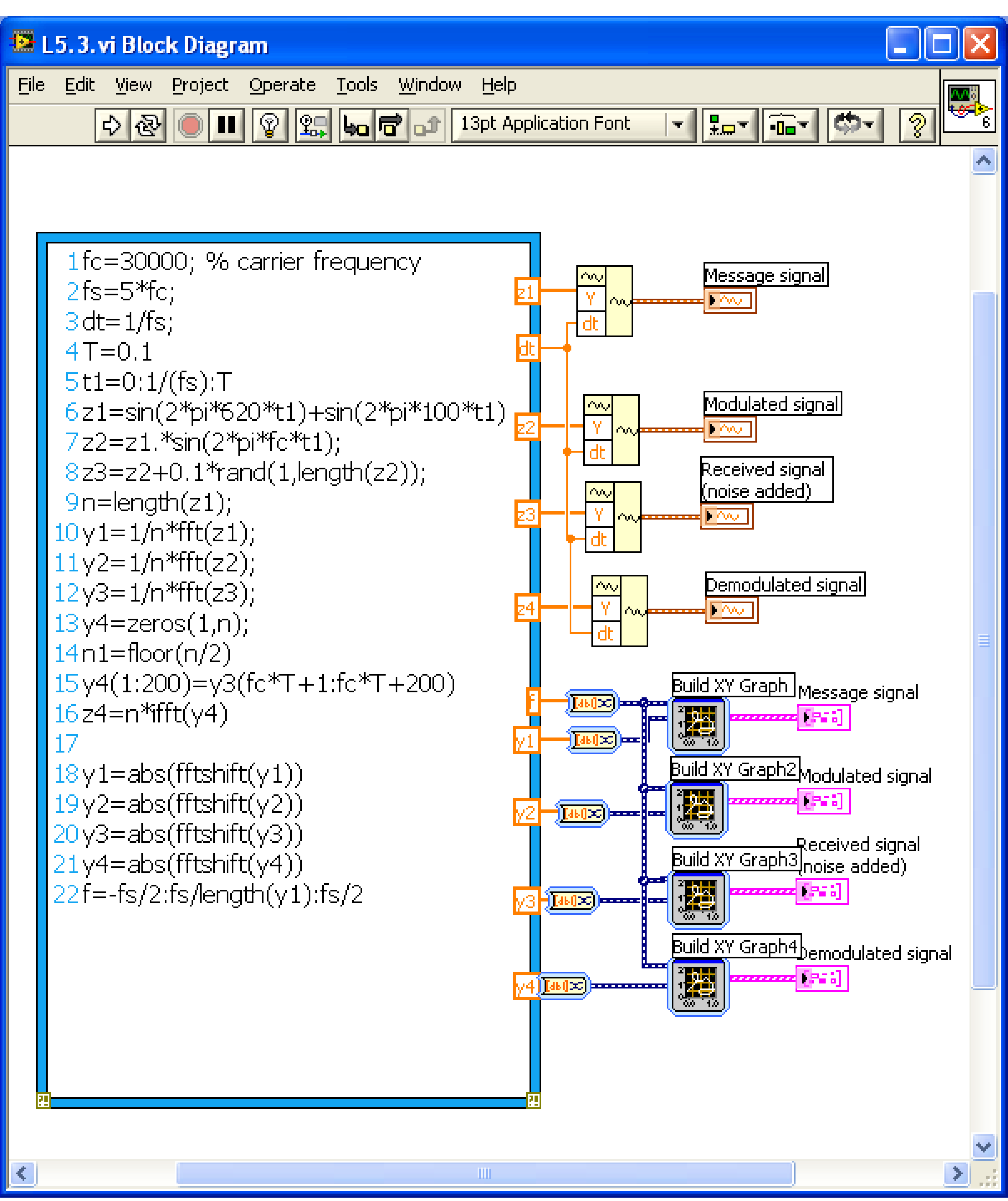
$$X(\omega) = \frac{1}{2}(X_m(\omega - \omega_c) + X_m(\omega + \omega_c))$$

At the receiver, some noisy version of this transmitted signal is received. The signal information resides in the envelope of the modulated signal, and thus an envelope detector can be used to recover the message signal.

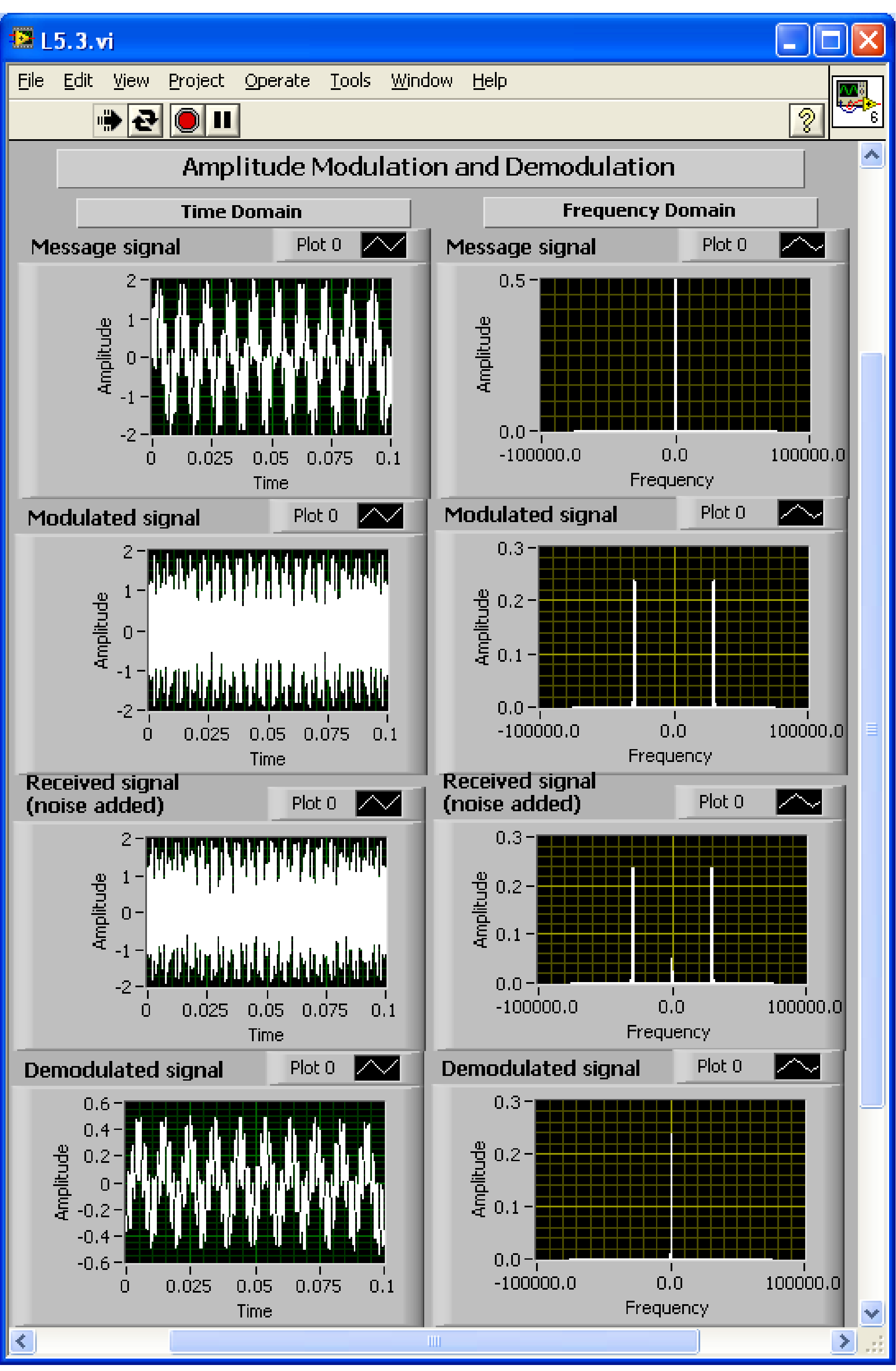
[link] shows the completed block diagram of the amplitude modulation and demodulation system. In this example, use the combination of two sine

waves to serve as a message signal. The signal is modulated with a high-frequency carrier, and some random noise is added. The frequency domain versions of the signals can also be observed using the function `fft`. As stated in Equation (3), the CTFT of the modulated signal is merely some frequency-shifted version of the original signal. In single sideband (SSB) modulation, only one side of the spectrum is transmitted due to symmetry. That is, just one side of the spectrum is taken and converted into a time signal using the function `ifft`.

[link] shows the completed front panel of this system. The Message signal, Modulated signal, Received signal (modulated signal with additional noise) and Demodulated signal are displayed in four waveform graphs in both the time and frequency domains.



Block Diagram of an Amplitude Modulation and Demodulation System



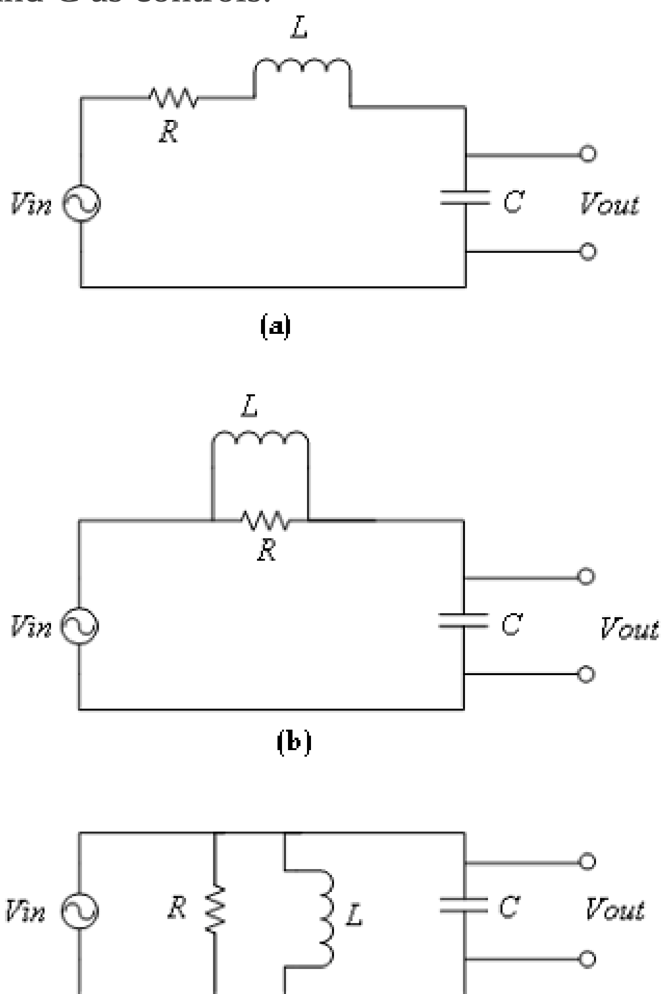
Front Panel of an Amplitude Modulation and Demodulation System

Lab Exercises

Exercise:

Problem: **Circuit Analysis**

Find and plot the frequency response (both magnitude and phase spectrum) of each of the circuits shown in [link]. Set the values of R, L and C as controls.



Linear RLC Circuits

Solution:

Insert Solution Text Here

Exercise:

Problem: Morse Coding

Consider a message containing some hidden information. Furthermore, to make it interesting, suppose the message contains a name. Assume that the message was coded using the amplitude modulation scheme as follows [link]:

Equation:

$$x(t) = x_{m1}(t)\cos(2\pi f_1 t) + x_{m2}(t)\cos(2\pi f_2 t) + x_{m3}(t)\cos(2\pi f_3 t)$$

where $x_{m1}(t)$,$x_{m2}(t)$ and $x_{m3}(t)$ are the (message) signals containing the three letters of the name. More specifically, each of the signals, $x_{m1}(t)$,$x_{m2}(t)$ and $x_{m3}(t)$, corresponds to a single letter of the alphabet. These letters are encoded using the International Morse Code as indicated below [7]:

A	. —	H	....	O	— — —	V	... —
B	—...	I	..	P	. — —.	W	. — —
C	—. — .	J	. — — —	Q	— — . —	X	—.. —
D	—..	K	—. —	R	. — .	Y	—. — —
E	.	L	. — ..	S	...	Z	— — ..
F	.. — .	M	— —	T	—		
G	— — .	N	—.	U	.. —		

Now to encode the letter A, one needs only a dot followed by a dash. That is, only two prototype signals are needed – one to represent the dash and one to represent the dot. Thus, for instance, to represent the letter A, set $x_{m1}(t) = d(t) + \text{dash}(t)$, where $d(t)$ represents the dot signal and $\text{dash}(t)$ the dash signal. Similarly, to represent the letter O, set $x_{m1}(t) = 3\text{dash}(t)$.

Find the prototype signals $d(t)$ and $\text{dash}(t)$ in the file morse.mat on the book website. After loading the file morse.mat

```
>>load morse
```

the signals $d(t)$ and $\text{dash}(t)$can be located in the vectors dot and dash, respectively. The hidden signal, which is encoded, per Equation (4), containing the letters of the name, is in the vector xt Let the three modulation frequencies f_1,$f_2$and f_3 be 20, 40 and 80 Hz, respectively.

• Using the amplitude modulation property of the CTFT, determine the three possible letters and the hidden name. (Hint: Plot the CTFT of xt Use the values of T and τau contained in the file.)

• Explain the strategy used to decode the message. Is the coding technique ambiguous? That is, is there a one-to-one mapping between the message waveforms ($x_{m1}(t)$,$x_{m2}(t)$,$x_{m3}(t)$)) and the alphabet letters? Or can you find multiple letters that correspond to the same message waveform?

Solution:

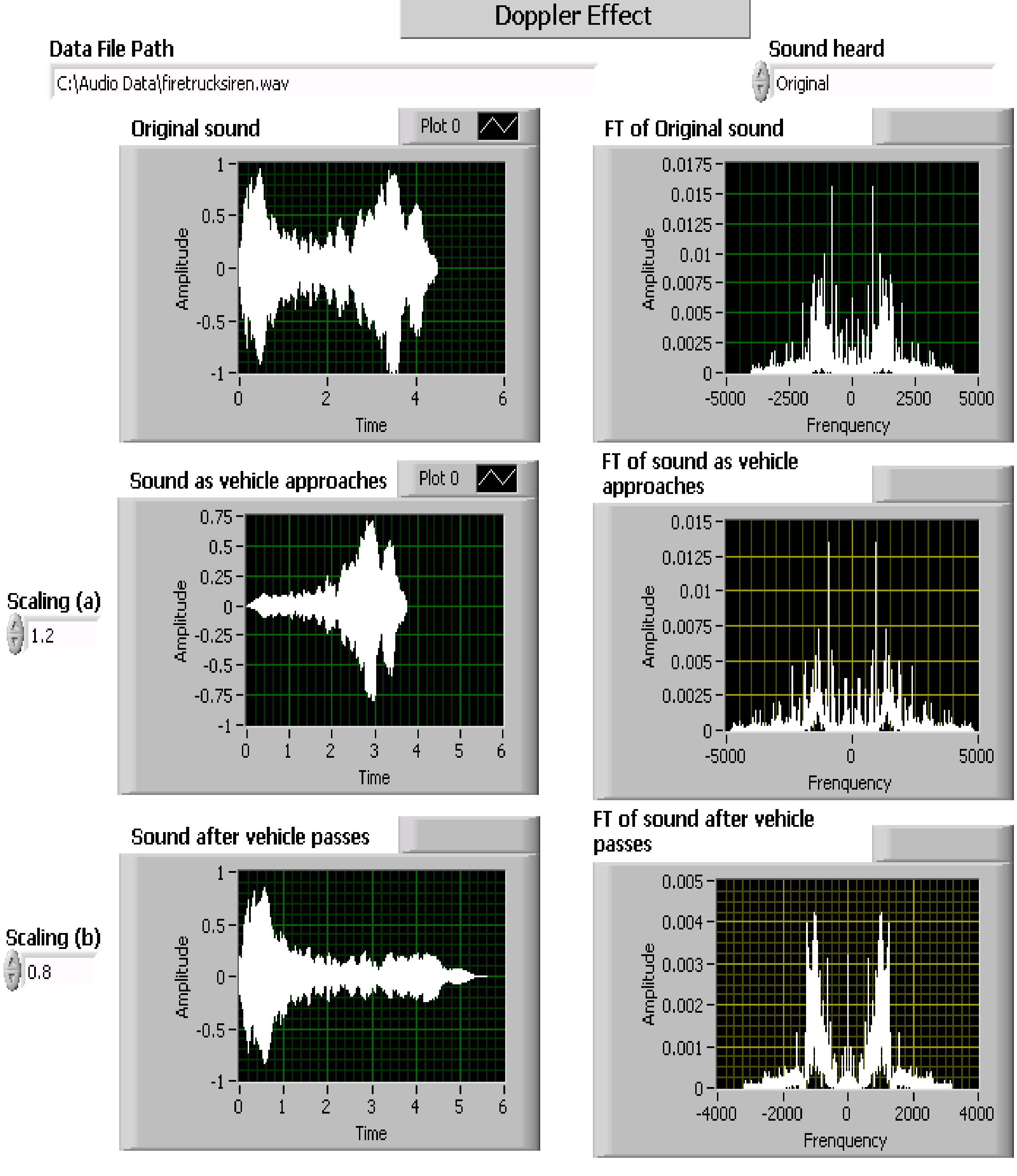
Insert Solution Text Here

Exercise:

Problem: Doppler Effect

The Doppler effect phenomenon was covered in a previous chapter. In this exercise, let us examine the Doppler effect with a real sound wave rather than a periodic signal. The wave file firetrucksiren.wav on the

book website contains a firetruck siren. Read the file using the LabVIEW MathScript function `wavread` and produce its upscale and downscale versions. Show the waves in the time and frequency domains (find the CTFT). Furthermore, play the sounds using the LabVIEW function **Play Waveform**. [link] shows a typical front panel for this system.



Front Panel of Doppler Effect System

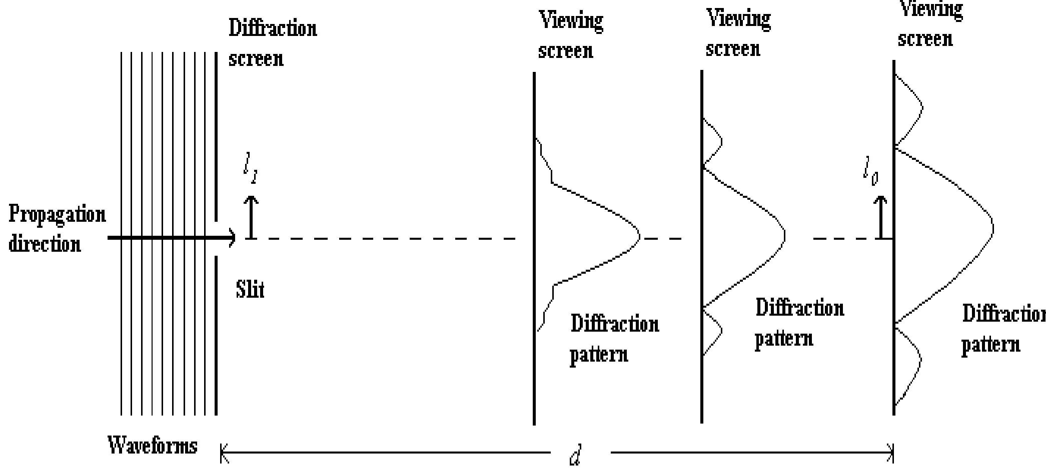
Solution:

Insert Solution Text Here

Exercise:

Problem: Diffraction of Light

The diffraction of light can be described as a Fourier transform [link]. Consider an opaque screen with a small slit being illuminated by a normally incident uniform light wave, as shown in [link].



Diffraction of Light

Considering that $d >> \pi {l_1}^2 / \lambda$provides a good approximation for any l_1 in the slit, the electric field strength of the light striking the viewing screen can be expressed as [link]

Equation:

$$E_0(l_0) = K \frac{e^{j(2\pi d/\lambda)}}{j\lambda d} e^{j(\pi/\lambda d){l_0}^2} \int_{-\infty}^{\infty} E_1(l_1) e^{-j(2\pi/\lambda d) l_0 l_1} \mathrm{dl}_1$$

where

E_1= field strength at diffraction screen

E_0 = field strength at viewing screen

K = constant of proportionality

λ= wavelength of light

The above integral is in fact Fourier transformation in a different notation. One can write the field strength at the viewing screen as [link]
Equation:

$$(t)$$

$$E_{1\,f \rightarrow l_0/\lambda d}$$

$$E_0(l_0) = K \frac{e^{j(2\pi d/\lambda)}}{j\lambda d} e^{j(\pi/\lambda d) l_0{}^2} \text{CTFT}$$

The intensity $I(l_0)$of the light at the viewing screen is the square of the magnitude of the field strength. That is,
Equation:

$$I(l_0) = \mid E_0(l_0) \mid^2$$

1. Plot the intensity of the light at the viewing screen. Set the slit width to this range (0.5 to 5 mm), the wavelength of light $\lambda$to this range (300 to 800 nm), and the distance of the viewing screen $d$to this range (10 to 200 m) as controls. Assume the constant of proportionality is 10^{-3}, and the electric field strength at the diffraction screen is 1 V/m.

2. Now replace the slit with two slits, each 0.1 mm in width, separated by 1 mm (center-to-center) and centered on the optical axis. Plot the intensity of light in the viewing screen by setting the parameters in part (1) as controls.

Solution:

Insert Solution Text Here

Digital Signals and Their Transforms

In this lab, we learn how to compute the continuous-time Fourier transform (CTFT), normally referred to as Fourier transform, numerically and examine its properties. Also, we explore noise cancellation and amplitude modulation as applications of Fourier transform.

In the previous labs, different mathematical transforms for processing analog or continuous-time signals were covered. Now let us explore the mathematical transforms for processing digital signals. Digital signals are sampled (discrete-time) and quantized version of analog signals. The conversion of analog-to-digital signals is implemented with an analog-to-digital (A/D) converter, and the conversion of digital-to-analog signals is implemented with a digital-to-analog (D/A) converter. In the first part of the lab, we learn how to choose an appropriate sampling frequency to achieve a proper analog-to-digital conversion. In the second part of the lab, we examine the A/D and D/A processes.

Sampling and Aliasing

Sampling is the process of generating discrete-time samples from an analog signal. First, it is helpful to mention the relationship between analog and digital frequencies. Consider an analog sinusoidal signal $x(t)=A\cos(\omega t+\varphi)$. Sampling this signal at $t=\mathrm{nT}_s$, with the sampling time interval of T_s, generates the discrete-time signal

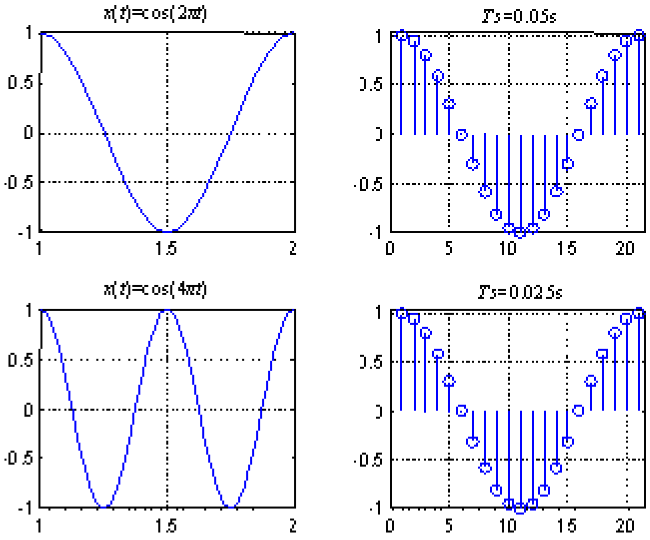
Equation:

$$x[n]=A\cos(\omega\mathrm{nT}_s+\varphi)=A\cos(\theta n+\varphi),\quad n=0{,}1{,}2{,}...,$$

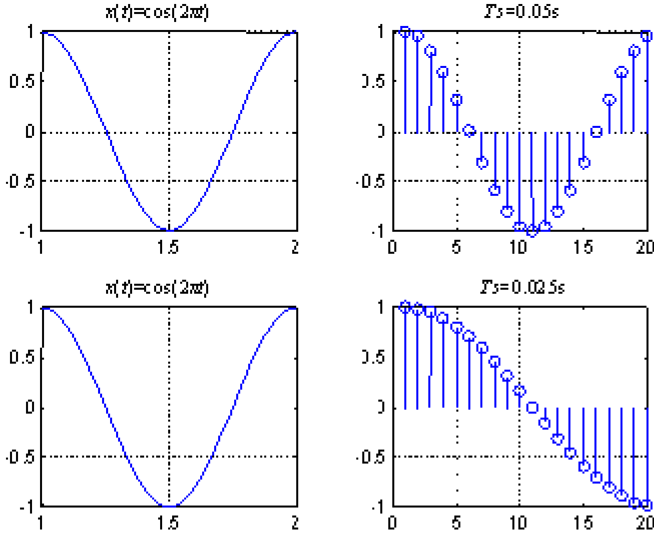
where $\theta=\omega T_s=\frac{2\pi\mathrm{f}}{f_s}$ denotes digital frequency with units being radians (as compared to analog frequency ω with units being radians/second).

The difference between analog and digital frequencies is more evident by observing that the same discrete-time signal is obtained from different continuous-time signals if the product ωT_s remains the same. (An example is shown in [link].) Likewise, different discrete-time signals are obtained

from the same analog or continuous-time signal when the sampling frequency is changed. (An example is shown in [link].) In other words, both the frequency of an analog signal f and the sampling frequency f_s define the digital frequency $\theta$of the corresponding digital signal.



Sampling of Two Different Analog Signals Leading to the Same Digital Signal

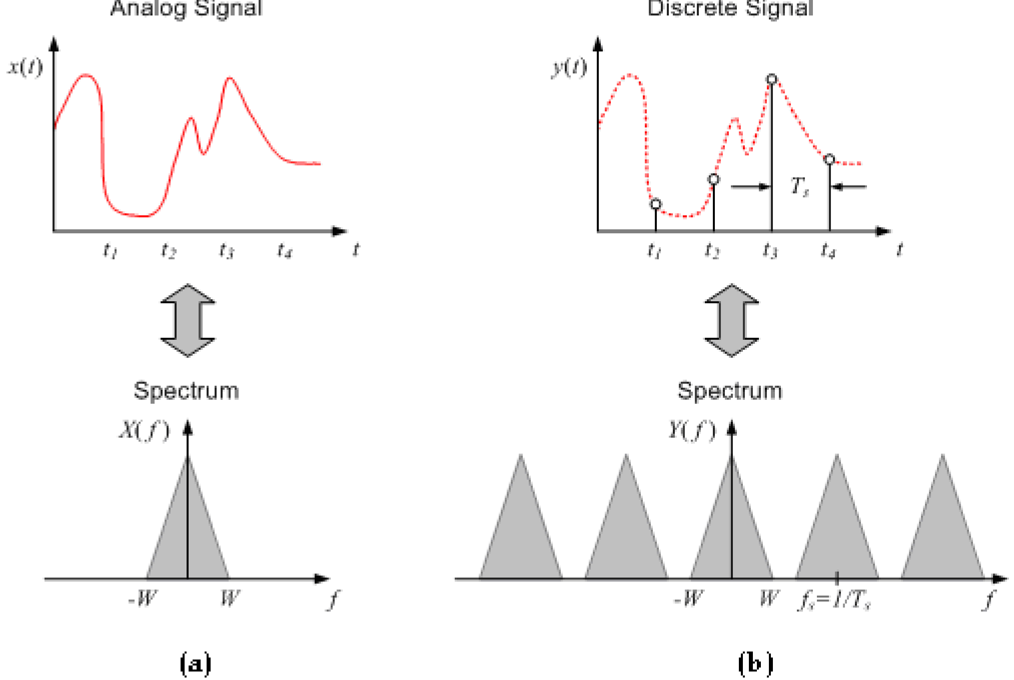


Sampling of the Same Analog Signal
Leading to Two Different Digital Signals

It helps to understand the constraints associated with the above sampling process by examining signals in the frequency domain. The Fourier transform pairs for analog and digital signals are stated as

Fourier transform pair for analog signals	$\begin{cases} X(j\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} \mathrm{dt} \\ x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(j\omega) e^{j\omega t} d\omega \end{cases}$
Fourier transform pair for discrete signals	$\begin{cases} X(e^{j\theta}) = \sum_{n=-\infty}^{\infty} x[n] e^{-\mathrm{j}n\theta}, \quad \theta = \omega T_s \\ x[n] = \frac{1}{2\pi} \int_{-\pi}^{\pi} X(e^{j\theta}) e^{\mathrm{j}n\theta} d\theta \end{cases}$

Fourier transform pairs for analog and digital signals

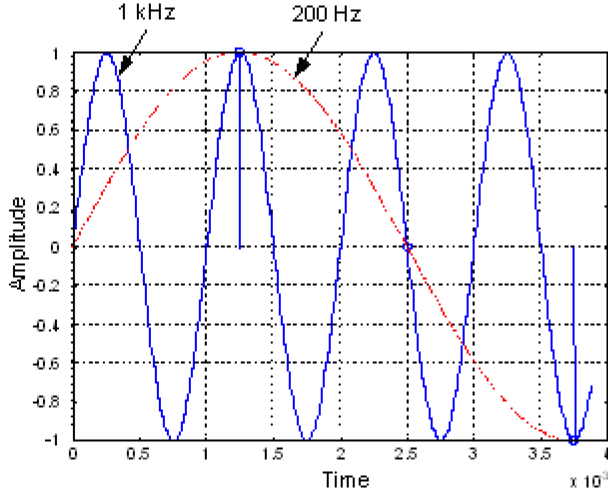


(a) Fourier Transform of a Continuous-Time Signal, (b) Its Discrete-Time Version

As illustrated in [link], when an analog signal with a maximum bandwidth of W(or a maximum frequency of $f_{\max}$) is sampled at a rate of $T_s = \frac{1}{f_s}$, its corresponding frequency response is repeated every 2π radians, or f_s. In other words, the Fourier transform in the digital domain becomes a periodic version of the Fourier transform in the analog domain. That is why, for discrete signals, one is interested only in the frequency range $[0, f_s/2]$.

Therefore, to avoid any aliasing or distortion of the discrete signal frequency content and to be able to recover or reconstruct the frequency content of the original analog signal, we must have $f_s \geq 2\mathrm{f}_{\max}$. This is known as the Nyquist rate. The sampling frequency should be at least twice the highest frequency in the analog signal. Normally, before any digital manipulation, a front-end anti-aliasing lowpass analog filter is used to limit the highest frequency of the analog signal.

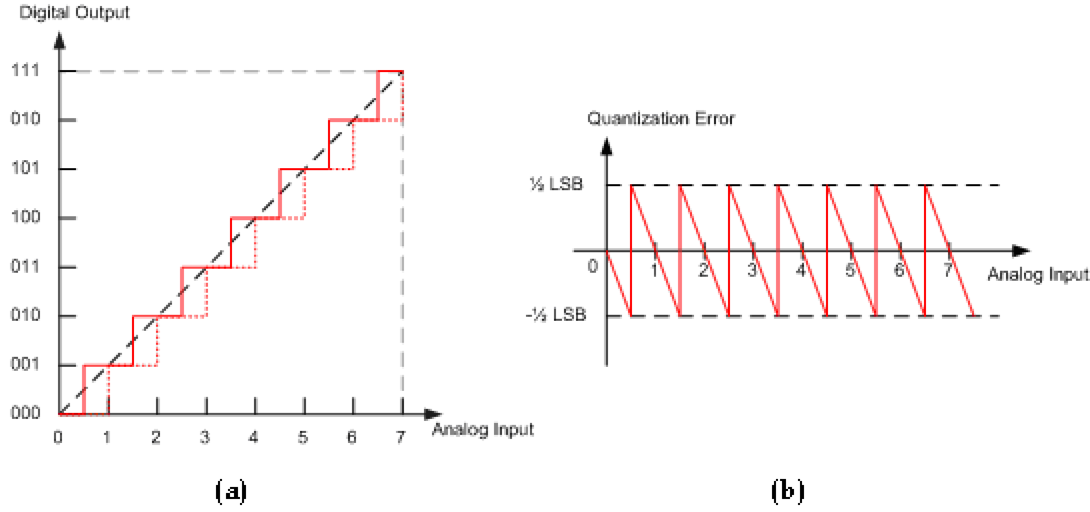
Let us further examine the aliasing problem by considering an undersampled sinusoid as depicted in [link] . In this figure, a 1 kHz sinusoid is sampled at $f_s = 0.8\text{kHz}$, which is less than the Nyquist rate of 2 kHz. The dashed-line signal is a 200 Hz sinusoid passing through the same sample points. Thus, at the sampling frequency of 0.8 kHz, the output of an A/D converter is the same if one uses the 1 kHz or 200 Hz sinusoid as the input signal. On the other hand, oversampling a signal provides a richer description than that of the signal sampled at the Nyquist rate.



Ambiguity Caused by Aliasing

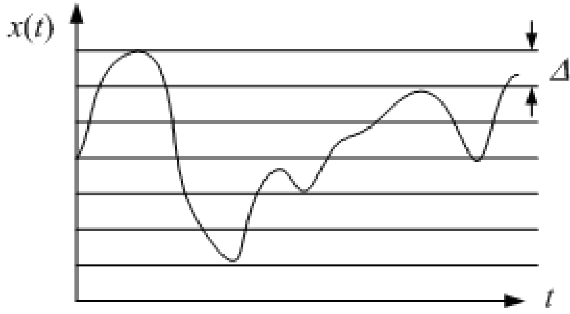
Quantization

An A/D converter has a finite number of bits (or resolution). As a result, continuous amplitude values get represented or approximated by discrete amplitude levels. The process of converting continuous into discrete amplitude levels is called quantization. This approximation leads to errors called quantization noise. The input/output characteristic of a 3-bit A/D converter is shown in [link] to illustrate how analog voltage values are approximated by discrete voltage levels.



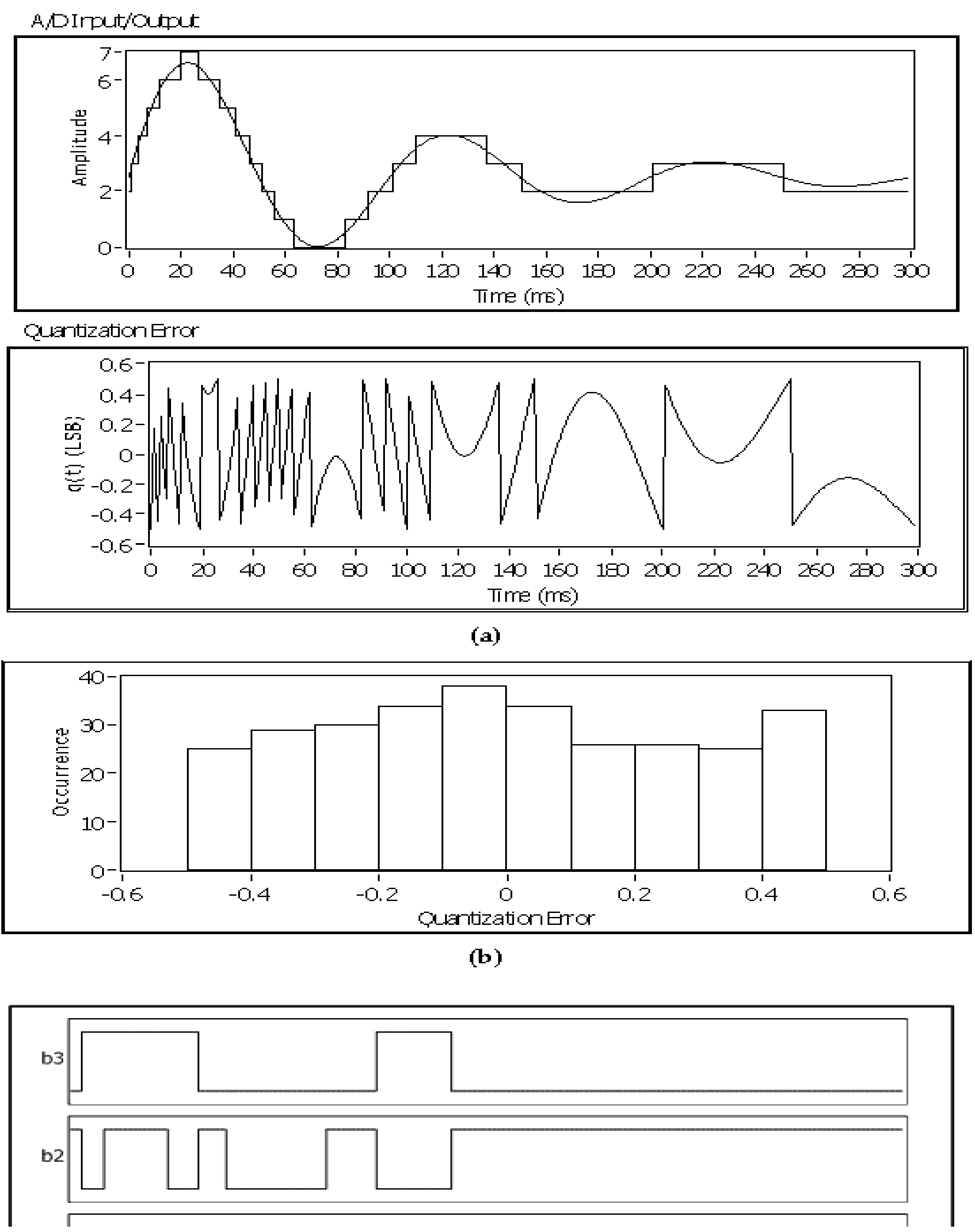
Characteristic of a 3-Bit A/D Converter: (a) Input/Output Transfer Function, (b) Additive Quantization Noise

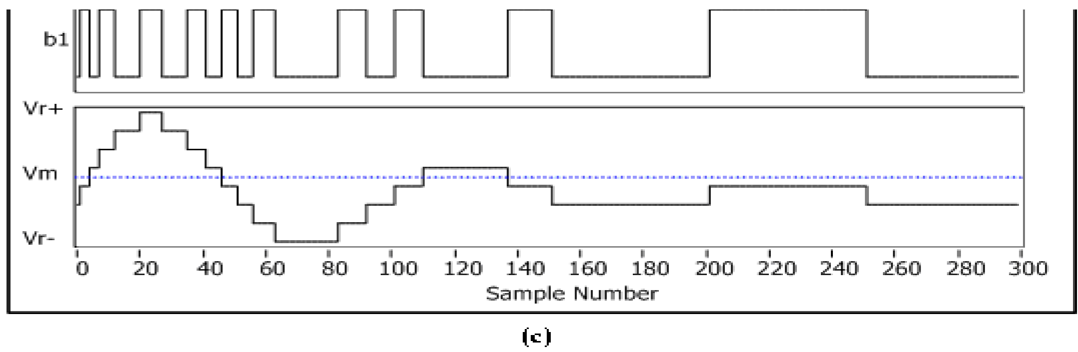
Quantization interval depends on the number of quantization or resolution levels, as illustrated in [link]. Clearly the amount of quantization noise generated by an A/D converter depends on the size of the quantization interval. More quantization bits translate into a narrower quantization interval and, hence, into a lower amount of quantization noise.



Quantization Levels

In [link], the spacing Δ between two consecutive quantization levels corresponds to one least significant bit (LSB). Usually, it is assumed that quantization noise is signal-independent and is uniformly distributed over –0.5 LSB and 0.5 LSB. [link] also shows the quantization noise of an analog signal quantized by a 3-bit A/D converter and the corresponding bit stream.





Quantization of an Analog Signal by a 3-Bit A/D Converter: (a) Output Signal and Quantization Error, (b) Histogram of Quantization Error, (c) Bit Stream

A/D and D/A Conversions

Because it is not possible to have an actual analog signal within a computer programming environment, an analog sinusoidal signal is often simulated by sampling it at a very high sampling frequency. Consider the following analog sine wave:
Equation:

$$x(t) = \cos(2\pi 1000t)$$

Sample this sine wave at 40 kHz to generate 0.125 seconds of $x(t)$. Note that the sampling interval, seconds, is very short, so $x(t)$ appears as an analog signal.

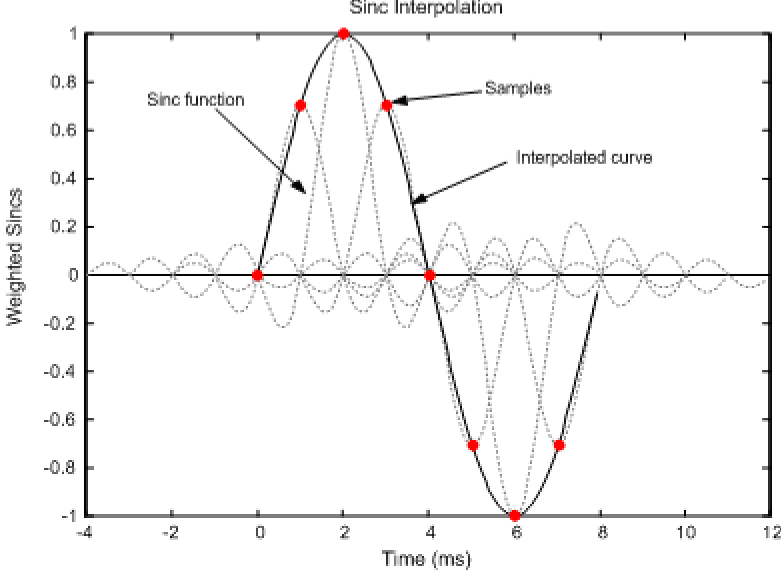
Sampling involves taking samples from an analog signal every seconds. The above example generates a discrete signal $x[n]$ by taking one sample from the analog signal every seconds. To get a digital signal, apply quantization to the discrete signal.

According to the Nyquist theorem, an analog signal z can be reconstructed from its samples by using the following equation:

Equation:

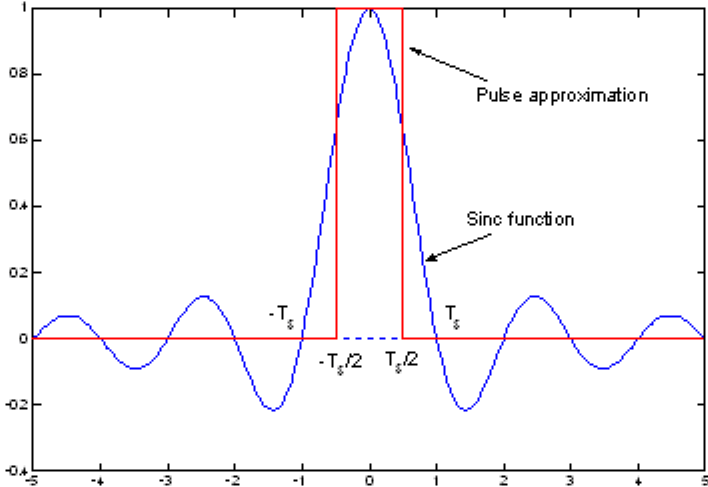
$$z(t) = \sum_{k=-\infty}^{\infty} z[\mathrm{kT}_s]\mathrm{sinc}(\frac{t - \mathrm{kT}_s}{T_s})$$

This reconstruction is based on the summations of shifted sinc (sinx/x) functions. [link] illustrates the reconstruction of a sine wave from its samples achieved in this manner.



Reconstruction of an Analog Sine Wave Based on its Samples, f= 125 Hz and f_s= 1 kHz

It is difficult to generate sinc functions by electronic circuitry. That is why, in practice, one uses an approximation of a sinc function. [link] shows an approximation of a sinc function by a pulse, which is easy to realize in electronic circuitry. In fact, the well-known sample and hold circuit performs this approximation.



Approximation of a Sinc Function by a Pulse

DTFT and DFT

Fourier transformation pairs for analog and discrete signals are expressed in [link]. Note that the discrete-time Fourier transform (DTFT) for discrete-time signals is the counterpart to the continuous-time Fourier transform (CTFT) for continuous-time signals. Also, the discrete Fourier transform (DFT) is the counterpart to the Fourier series (FS) for continuous-time signals as shown in [link] . [link] shows a list of these transformations and their behavior in the time and frequency domains.

Fourier series for periodic	$\begin{cases} X_k = \frac{1}{T}\int_{-T/2}^{T/2} x(t) e^{-j\omega_0 \mathrm{kt}} \mathrm{dt} \\ x(t) = \sum_{k=-\infty}^{\infty} X_k e^{j\omega_0 \mathrm{kt}} \end{cases}$, where T denotes period and ω_0 fundamental frequency

analog signals	
Discrete Fourier transform (DFT) for periodic discrete signals	$\begin{cases} X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}\text{nk}}, & k = 0,1,\dots,N-1 \\ x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi}{N}\text{nk}}, & n = 0,1,\dots,N-1 \end{cases}$

Fourier series pairs for analog and digital signals

Time domain	**Spectrum characteristics**	**Transformation type**
Continuous (periodic)	Discrete	FS
Continuous (aperiodic)	Continuous	CTFT
Discrete (periodic)	Discrete (periodic)	DFT
Discrete (aperiodic)	Continuous (periodic)	DTFT

Different Transformations for Continuous and Discrete Signals

Lab 6: Analog-to-Digital Conversion, DTFT and DFT

Sampling, Aliasing, Quantization and Reconstruction

The example in this section addresses sampling, quantization, aliasing and signal reconstruction concepts. [link] shows the completed block diagram of this example, where the following four control parameters are linked to a LabVIEW MathScript node:

Amplitude – to control the amplitude of an input sine wave

Phase – to control the phase of the input signal

Frequency – to control the frequency of the input signal

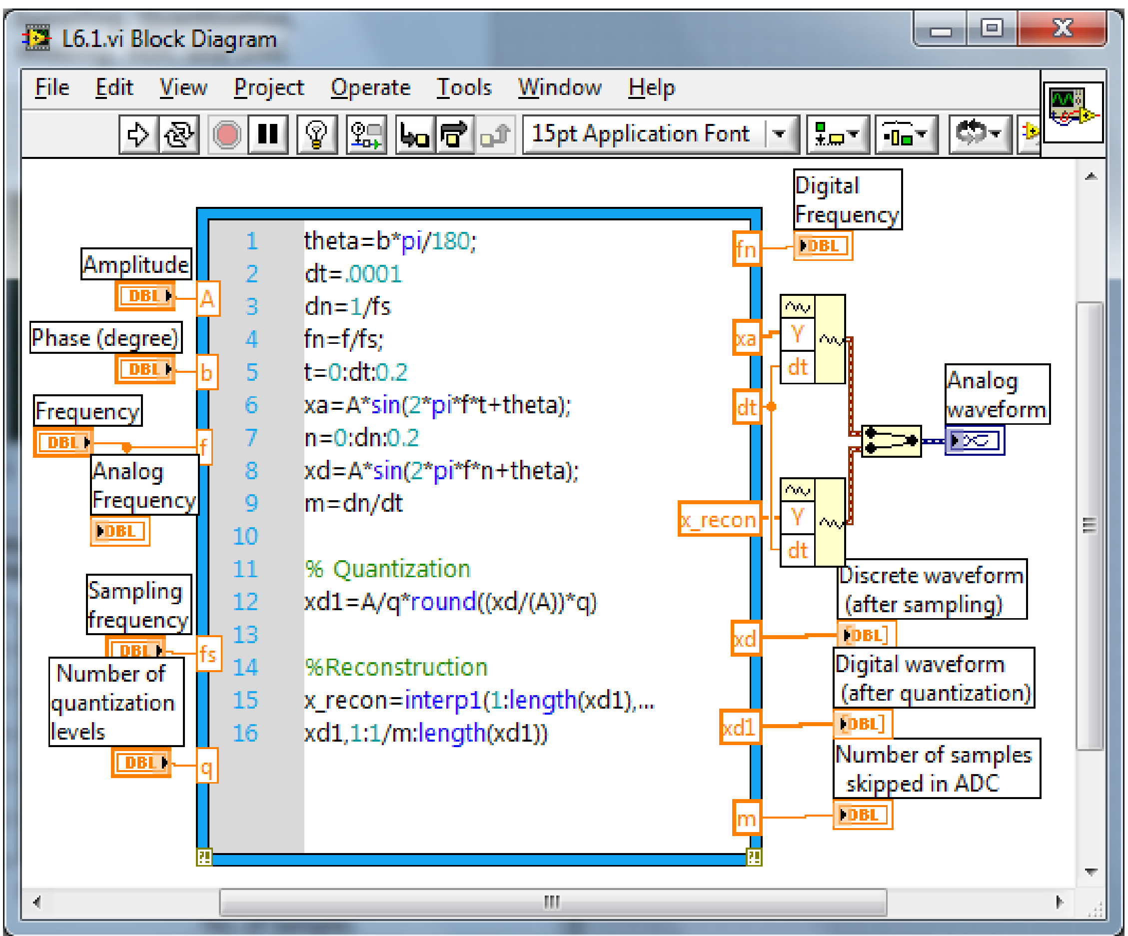
Sampling frequency – to control the sampling rate of the corresponding discrete signal

Number of quantization levels – to control the number of quantization levels of the corresponding digital signal

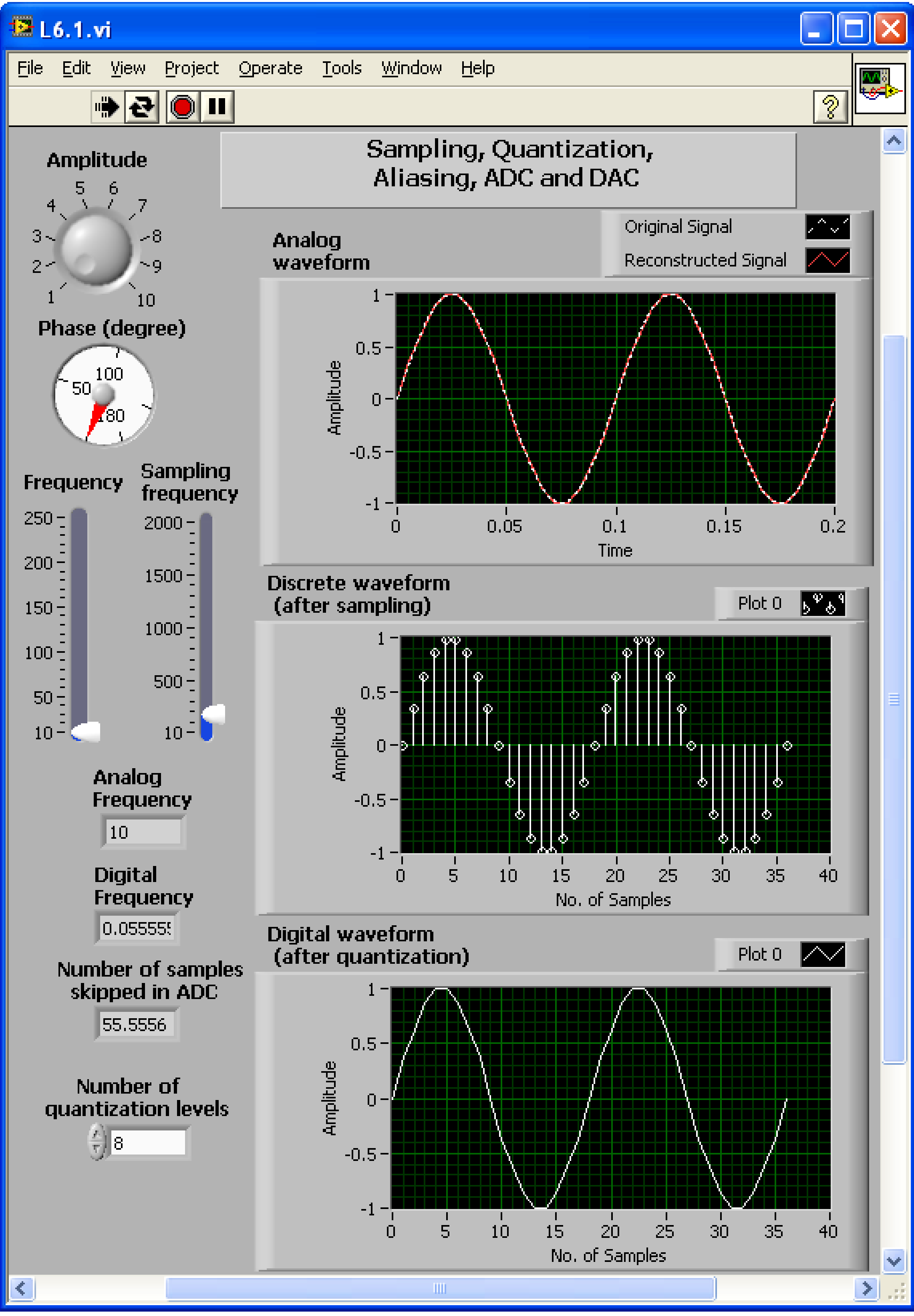
To simulate the analog signal via a .m file, consider a very small value of time increment dt (dt = 0.001). To create a discrete signal, sample the analog signal at a rate controlled by the sampling frequency. To simulate the analog signal, use the textual statement `xa=sin(2*pi*f*t)`, where t is a vector with increment dt = 0.001. To simulate the discrete signal, use the textual statement `xd=sin(2*pi*f*n)`, where n is a vector with increment dn. The ratio dn/dt indicates the number of samples skipped during the sampling process. Again, the ratio of analog frequency to sampling frequency is known as digital or normalized frequency. To convert the discrete signal into a digital one, perform quantization using the LabVIEW MathScript function `round`. Set the number of quantization levels as a control.

To reconstruct the analog signal from the digital one, use a linear interpolation technique via the LabVIEW MathScript function `interp1`. The samples skipped during the sampling process can be recovered after the

interpolation. Finally, display the Original signal and the Reconstructed signal in the same graph using the functions Build Waveform, Merge Signal and Waveform Graph. Discrete waveform, Digital waveform, Analog frequency, Digital frequency and Number of samples skipped in ADC are also included in the front panel, shown in [link]. Use this VI to examine proper signal sampling and reconstruction.



Block Diagram of Sampling, Aliasing, Quantization and Reconstruction



L6.1.vi
File Edit View Project Operate Tools Window Help
Sampling, Quantization, Aliasing, ADC and DAC
Amplitude
Phase (degree)
Frequency
Sampling frequency
Analog Frequency
10
Digital Frequency
0.05555
Number of samples skipped in ADC
55.5556
Number of quantization levels
8
Analog waveform
Original Signal
Reconstructed Signal
Amplitude
Time
Discrete waveform (after sampling)
Plot 0
No. of Samples
Digital waveform (after quantization)

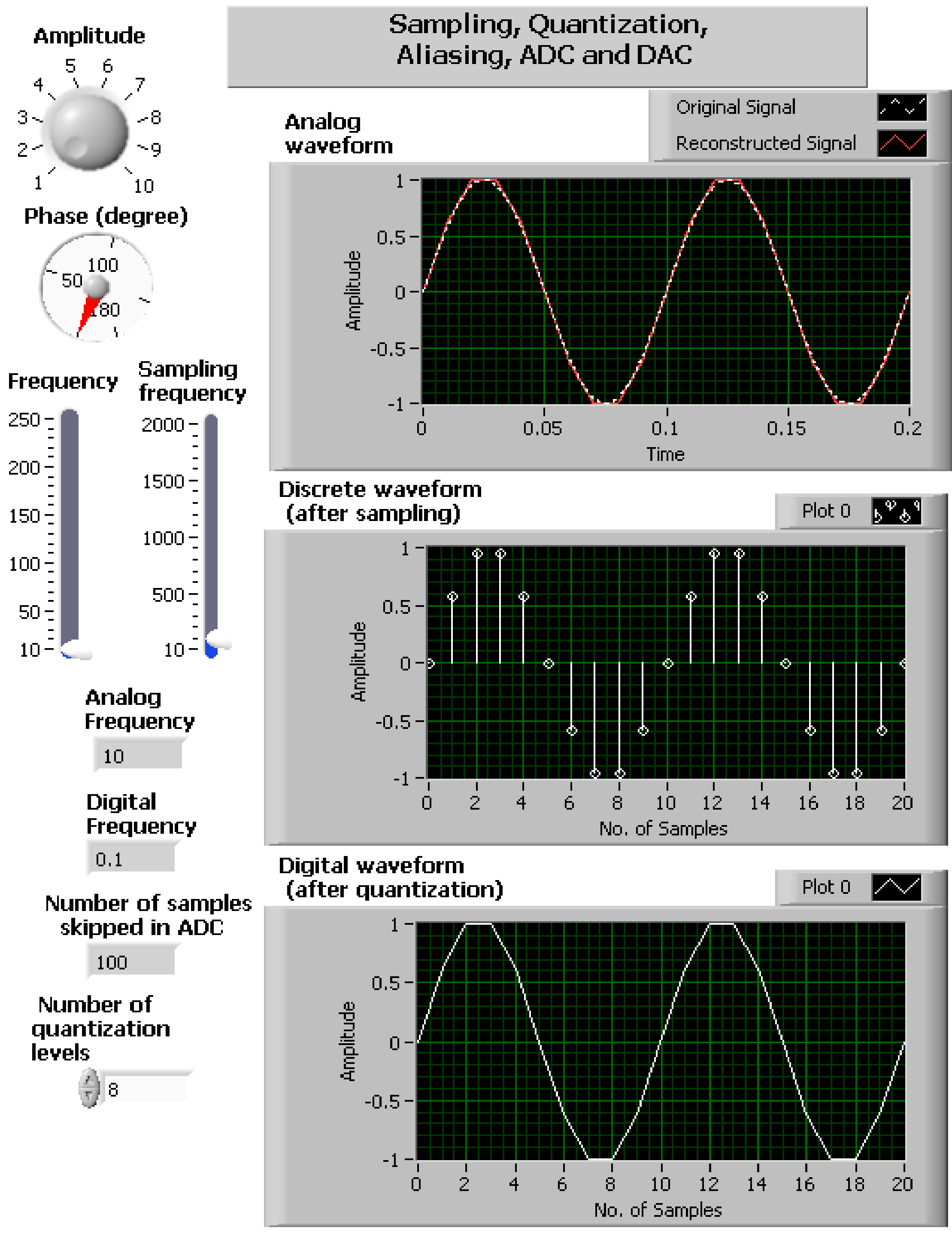
Front Panel of Sampling, Aliasing, Quantization and Reconstruction

Analog and Digital Frequency

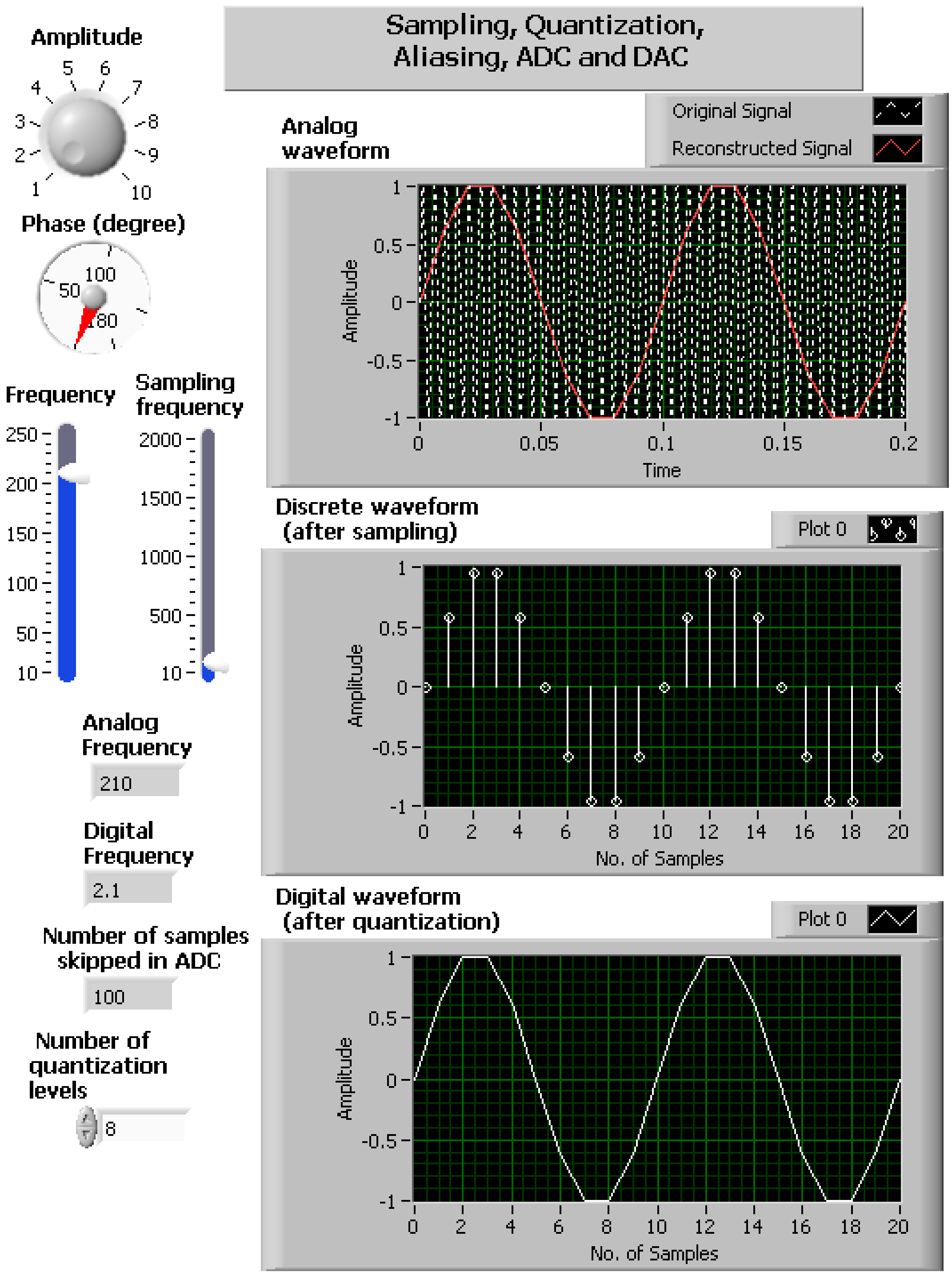
Digital frequency (θ) is related to analog frequency (f) via the sampling frequency, that is, $\theta = \frac{2\pi f}{f_s}$. Therefore, one can choose the sampling frequency (f_s) to increase the digital or normalized frequency of an analog signal by lowering the number of samples.

Aliasing

Set the sampling frequency to $f_s = 100\text{Hz}$ and change the analog frequency of the signal. Observe the output for $f_s = 10\text{Hz}$ and $f_s = 210\text{Hz}$ (See [link] and [link]). The analog signals appear entirely different in these two cases but the discrete signals are similar. For the second case, the sampling frequency is less than twice that of the analog signal frequency. This violates the Nyquist sampling rate leading to aliasing, which means one does not know from which analog signal the digital signal is created. Note the value of digital frequency is 0.1 radians for the first case and 2.1 radians for the second case. To prevent any aliasing, keep the digital frequency less than 0.5 radians.



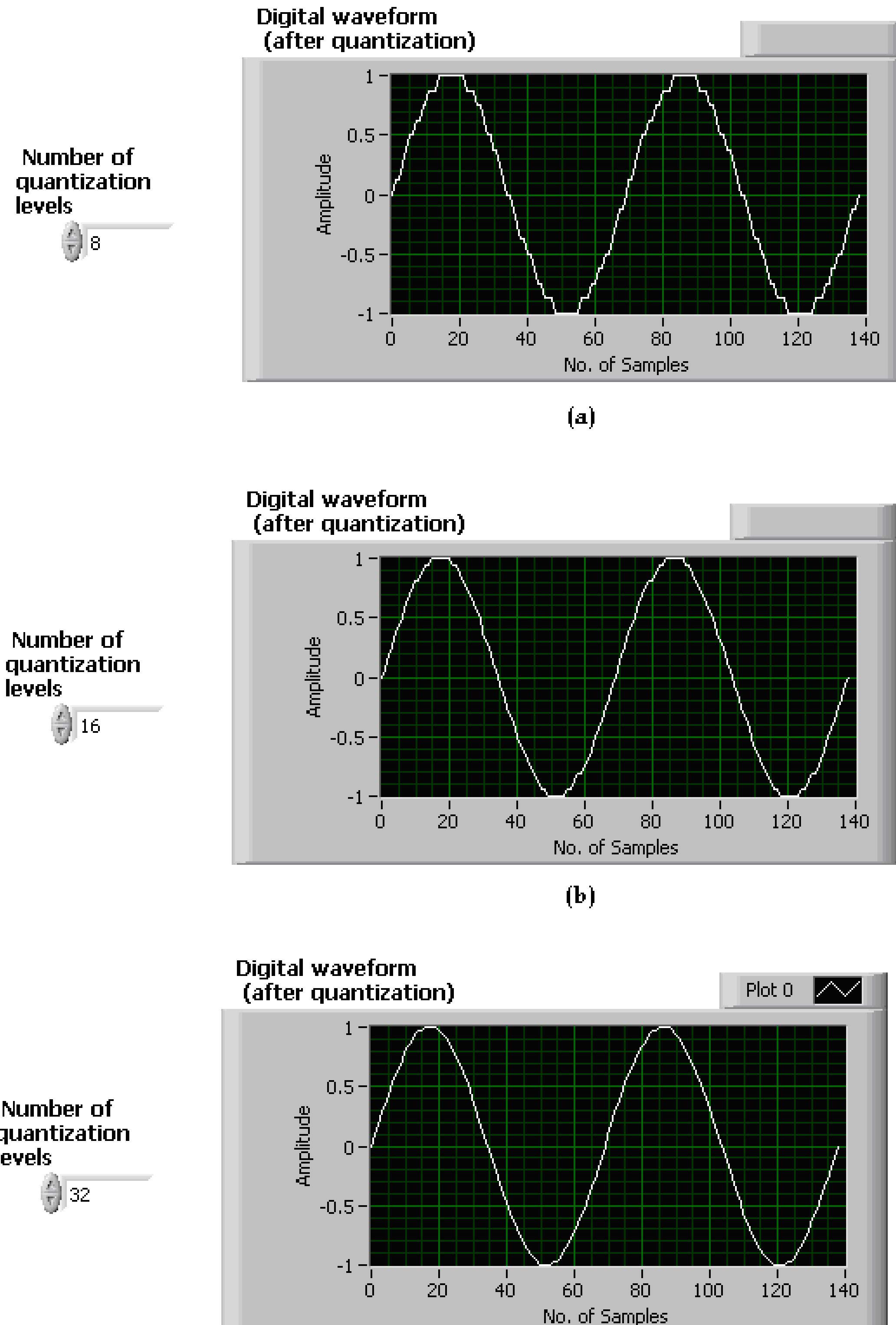
Analog and Discrete Waveforms with fs = 100 Hz and f = 10 Hz



Analog and Discrete Waveforms with fs = 100 Hz and f = 210 Hz

Quantization

Now change the Number of quantization levels for some fixed values of Frequency and Sampling Frequency. As the number of quantization levels is increased, the Digital waveform becomes smoother and a smaller amount of quantization error or noise is generated.

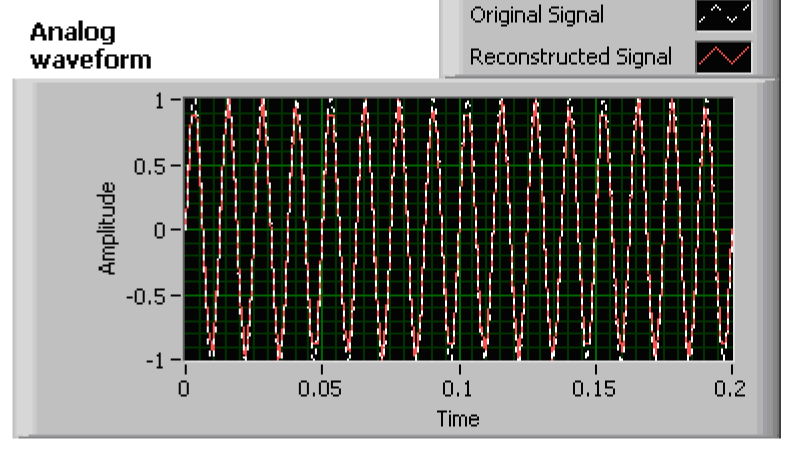


Digital Waveform with Different Numbers of Quantization

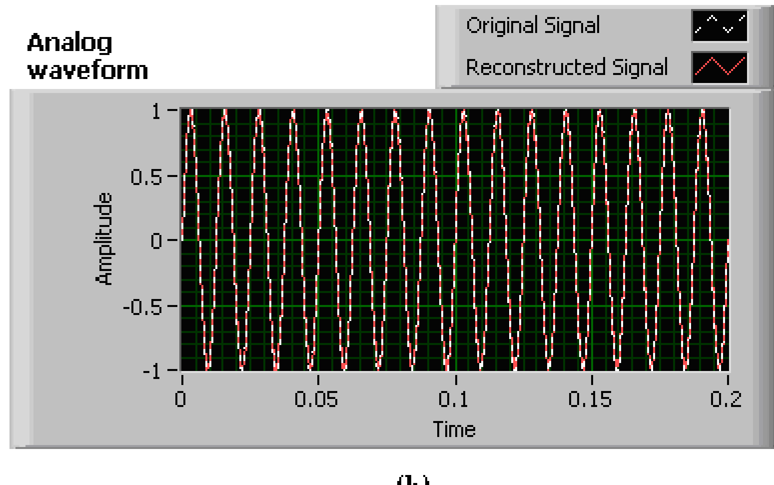
Levels: (a) 8, (b) 6, (c) 32

Signal Reconstruction

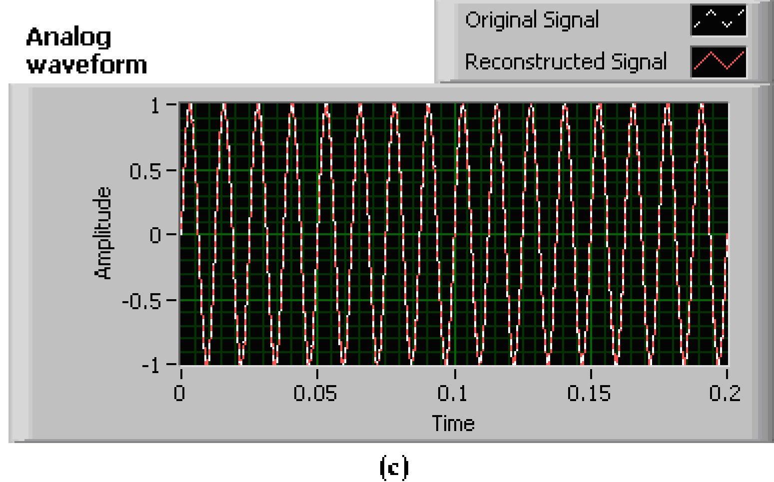
Next, set the frequency $f = 100\text{Hz}$ and vary the sampling frequency. Observe the reconstructed waveform. [link] shows the reconstructed signals for three different values of skipped samples. If the sampling frequency is increased, fewer samples are skipped during the analog-to-digital conversion, which makes the reconstruction process more accurate.



(a)



(b)



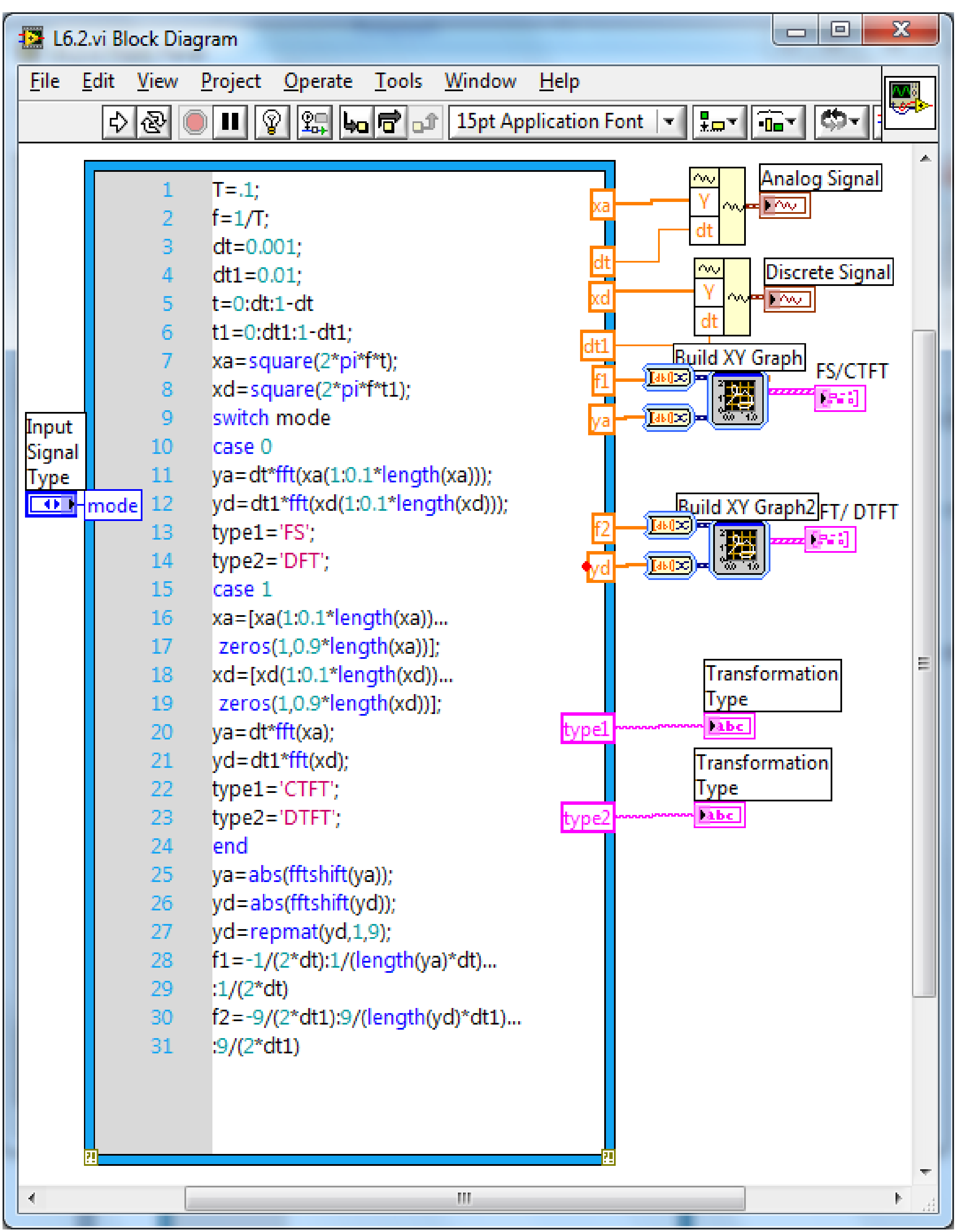
(c)

Signal Reconstruction with Different Number of Samples Skipped in ADC: (a) 20,

(b)10, (c) 5

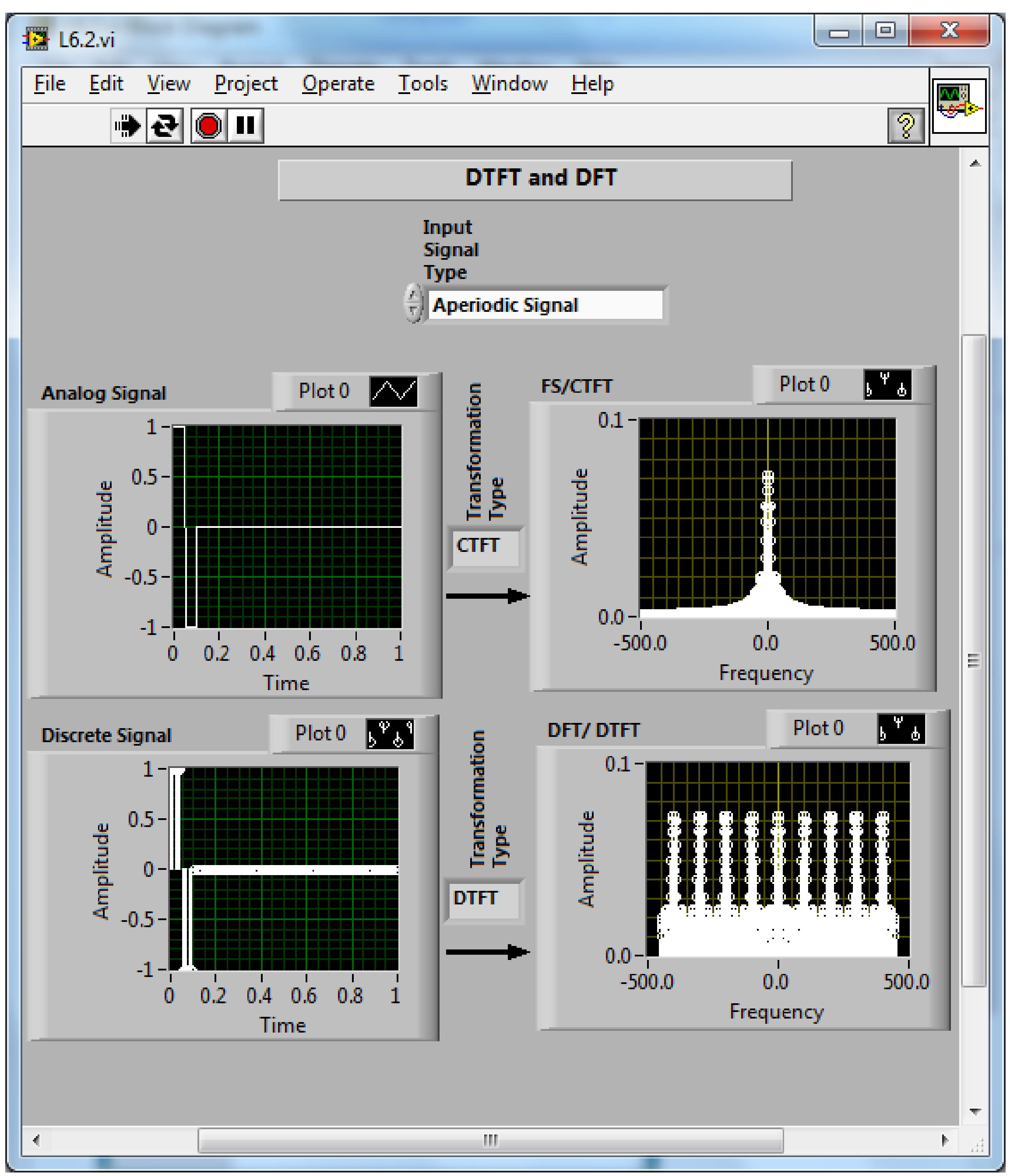
DTFT and DFT

In this example, let us compute and compare the DTFT and DFT of digital signals with the CTFT and FS of analog signals. [link] illustrates the completed block diagram of this transform comparison system. As discussed previously, to simulate an analog signal, consider a small time interval $(\mathrm{dt} = 0.001)$. The corresponding discrete signal is considered to be the same signal with a larger time interval $(\mathrm{dt1} = 0.01)$.

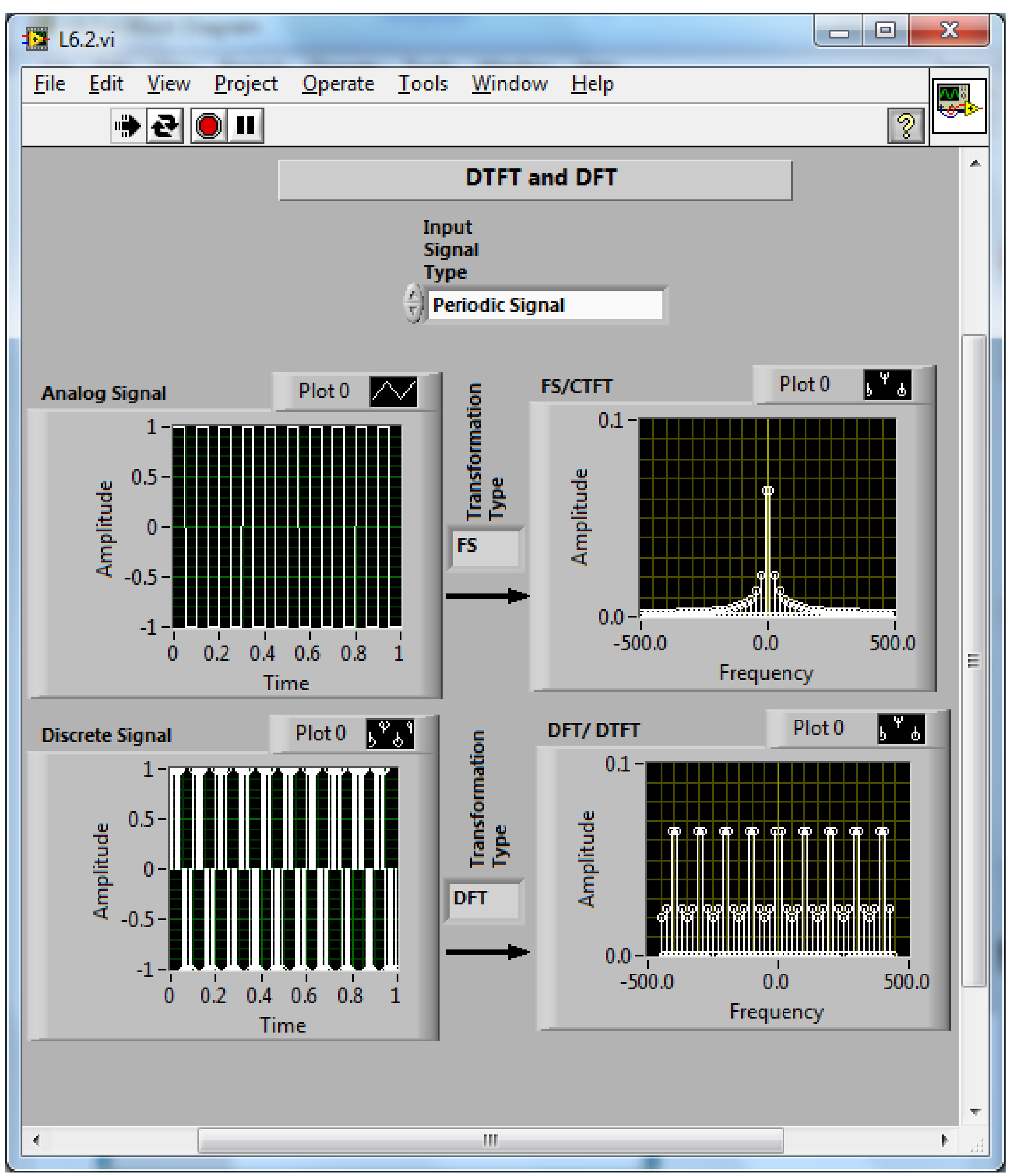


Block Diagram of a DTFT and DFT Transformation System

Generate a periodic square wave with the time period $T = 0.1$. Connect the input variable mode to an Enum Control to make the signal periodic or aperiodic. If the signal is periodic (case 0), compute the FS of the analog signal and the DFT of the digital signal using the `fft` function over one period of the signal. For aperiodic signals, only one period of the square wave is considered and the remaining portion is padded with zeros. For aperiodic signals, the transformations are CTFT (for analog signals) and DTFT (for digital signals), which are computed using the `fft` function. In fact, this function provides a computationally efficient implementation of the DFT transformation for periodic discrete-time signals. However, because simulated analog signals are actually discrete with a small time interval, this function is also used to compute the Fourier series for continuous-time signals. Because DFT requires periodicity, one needs to treat aperiodic signals as periodic with a period $T = \infty$to apply this useful function. That is why the `fft` function is also used for aperiodic signals to compute CTFT and DTFT (as done in the earlier labs). However, in practice, it should be noted that the period of the zero padded signal is not infinite but assumed long enough to obtain a close approximation. Apply the same approach to the computation of CTFT and DTFT. Because DTFT is periodic in the frequency domain, for digital signals, repeat the frequency representation using the textual statement `yd=repmat(yd,1,9)`, noting that the `fft` function computes the transformation for one period only.



Front Panel of a DTFT and DFT Transformation System: Aperiodic Signal



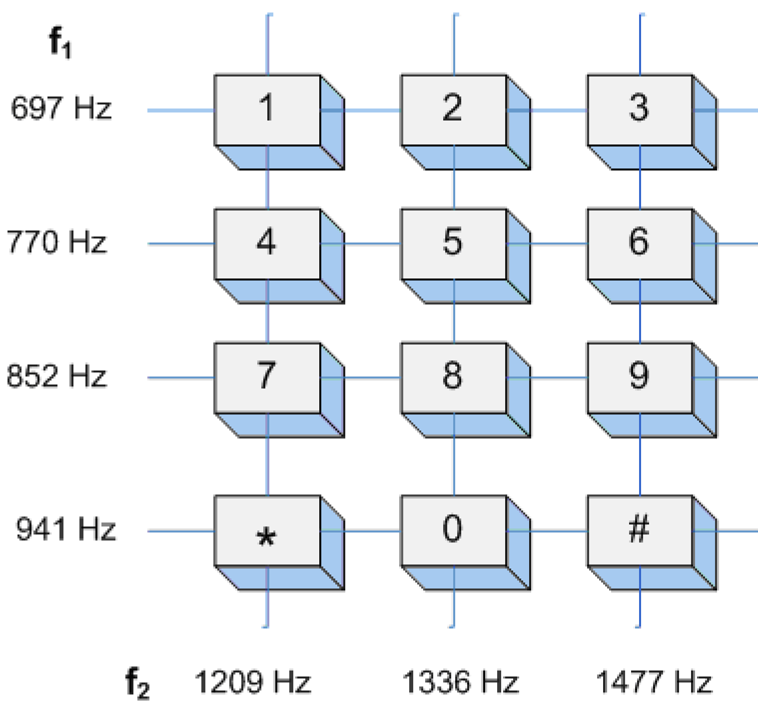
Front Panel of a DTFT and DFT Transformation System: Periodic Signal

[link] and [link] illustrates the front panel of the above transformation system. It shows the Analog signal and Discrete signal in the time and

frequency domains using two waveform graphs. The transformation type is also shown in the front panel for both of the signals.

Telephone Signal

Now let us examine a DFT application. In a touch-tone dialing system, the pressing of each button generates a unique set of two-tone signals, called dual-tone multi-frequency (DTMF) signals. A telephone central office processes these signals to identify the number a user presses. The tone frequency assignments for touch-tone dialing are shown in [link].



Frequency Assignments for Touch-Tone Dialing

The sound heard when a key is pressed is a signal composed of two sine waves. That is

Equation:

$$x(t) = \sin(2\pi f_1 t) + \sin(2\pi f_2 t)$$

For example, when a caller presses 1, the corresponding signal is

Equation:

$$x_1(t) = \sin(2\pi 697 t) + \sin(2\pi 1209 t)$$

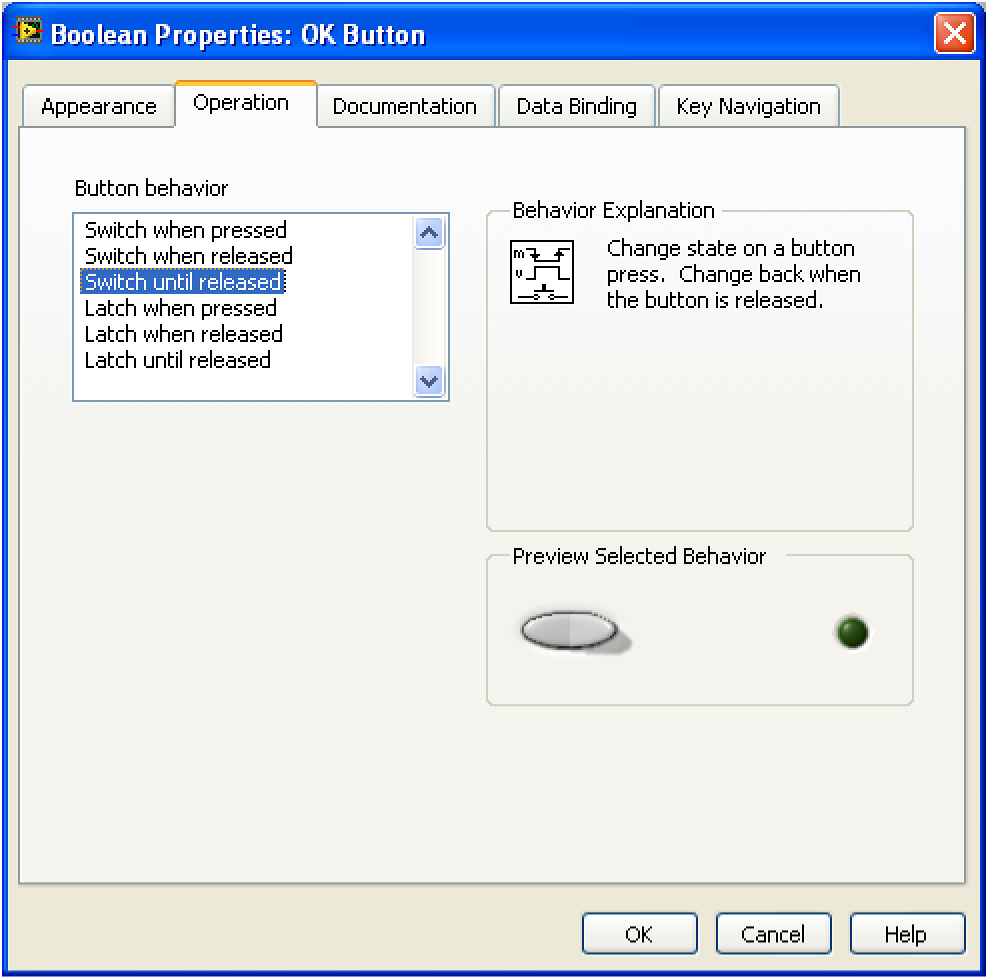
Other than touch-tone signals, modern telephone systems use DTMF event signals for dial tone, busy tone and ringing tone. Table 1 lists the frequency and timing for standard DTMF event signals.

Tone type	Frequency	Timing
Dial tone	350 and 440 Hz	Continuous
Ringing tone	480 and 620 Hz	Repeating cycles of 2 s on, 4 s off
Busy tone	480 and 620 Hz	0.5 s on, 0.5 s off

DTMF Event Signals

In this application, let us examine the touch-tone dialing system of a digital telephone. Ten input variables $(k_0, k_1, \ldots\ldots, k_9)$are assigned to the telephone keys (0,1,.....,9). Each input is connected to a Boolean control. Different types of Boolean controls can be created on the front panel. For this application, use **OK Buttons(Controls → Modern → Boolean → OK)**, each of which is marked with a number from 0 to 9. The selected operation properties of an **OK Button** is “Switch until released,” as shown in [link].

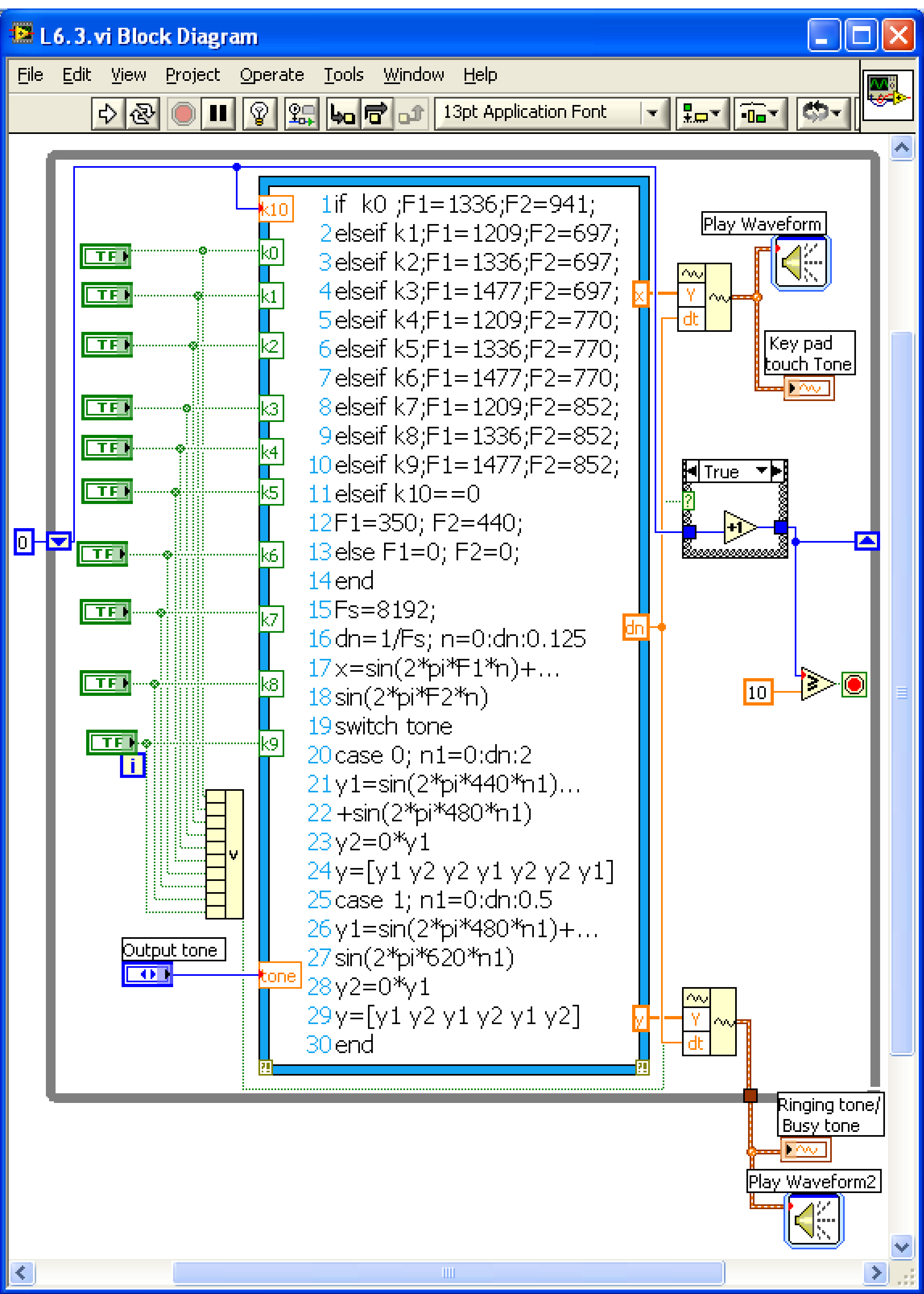
This ensures that the corresponding signal gets generated when a key is pressed and gets back into its initial position when the key is released.



Operation Properties of an OK Button

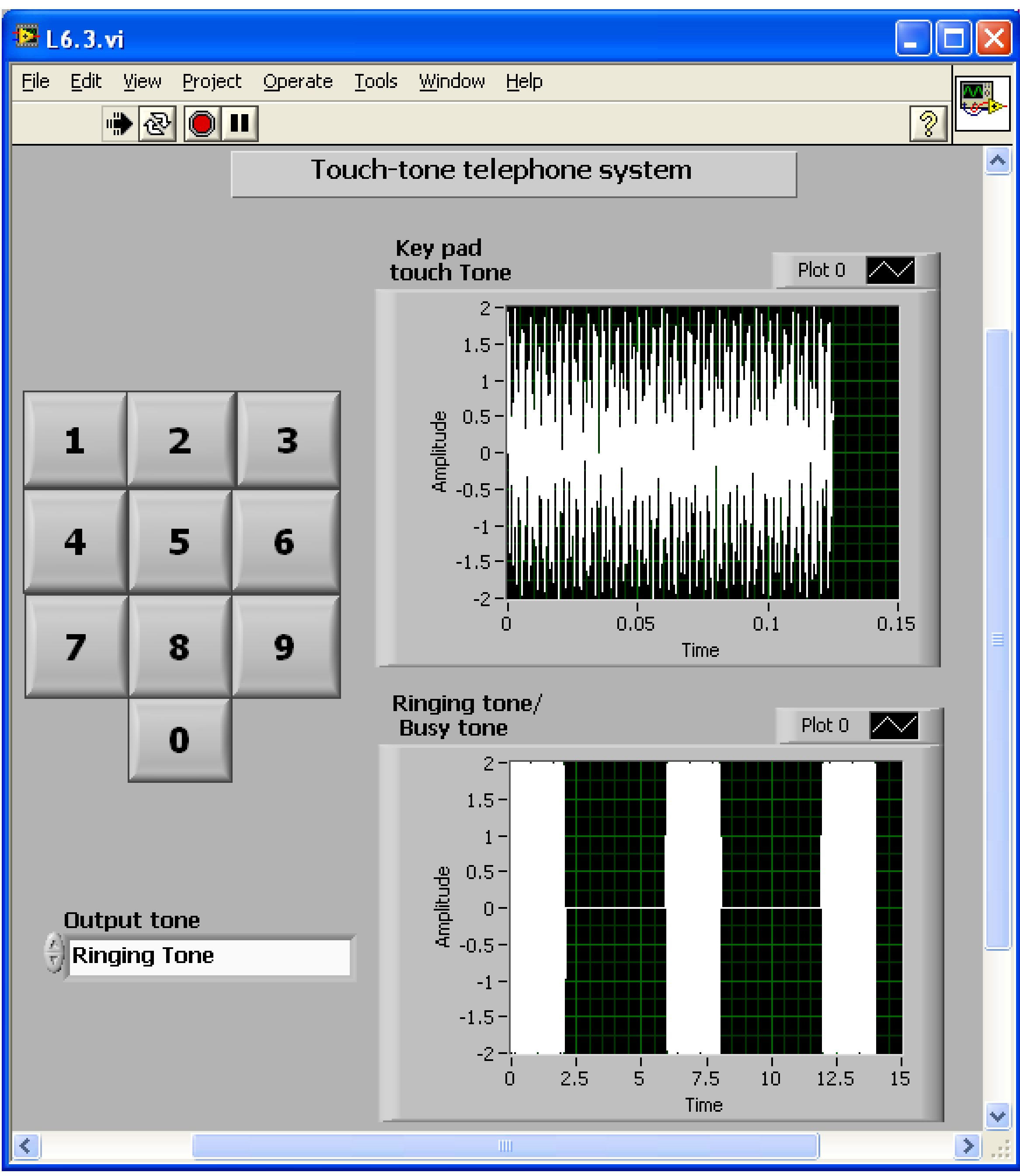
Use another input variable, k_{10}, to act as a counter to count the number of times the keys are pressed. At the beginning, when no key is pressed, the value of $k_{10}$is zero and the system returns the dial tone (350 and 440 Hz continuous tone). When the value of $k_{10}$is equal to 10, meaning that the keys were pressed for a total of 10 times, the system assumes that a valid phone number is dialed and returns the busy tone or ringing tone. Connect

all the Boolean inputs $(k_0, k_1, \ldots\ldots, k_9)$ to a **Compound Arithmatic** function **(Functions → Programming → Boolean → Compound Arithmatic)** and select the OR mode. The output of this function becomes true(1) if any number key is pressed. The result is connected to the Case selector input of a case structure. The input variable is also connected to the case structure. For true case, $k_{10}$is connected to an **Increment** function **(Functions→Programming →Numeric →Increment)** and for the false case, it is kept unchanged. The entire system is wrapped inside a **While Loop(Functions →Programming →Structures →While Loop)**. The output of the case structure is then connected to a **Greater or Equal** function **(Functions →Programming →Comparison →Greater or Equal)** to ensure that the program exits from the while loop when is greater than or equal to 10. The system shows the ringing tone or busy tone in the graph and plays the waveform. [link] shows the completed block diagram of the touch-tone telephone system.



Block Diagram of Touch-Tone Telephone System

[link] shows the front panel of the touch-tone telephone system. When the program is run, one can hear the dial tone and see the signal displayed in the upper waveform graph. As soon as any number key is pressed, the dial tone is stopped and the corresponding key pad tone is heard and displayed. When keys are pressed 10 times (a valid phone number), the system plays the ringing tone or busy tone depending on the setting and displays the tone in the lower waveform graph.



Front Panel of a Touch-Tone Telephone System

Lab Exercises

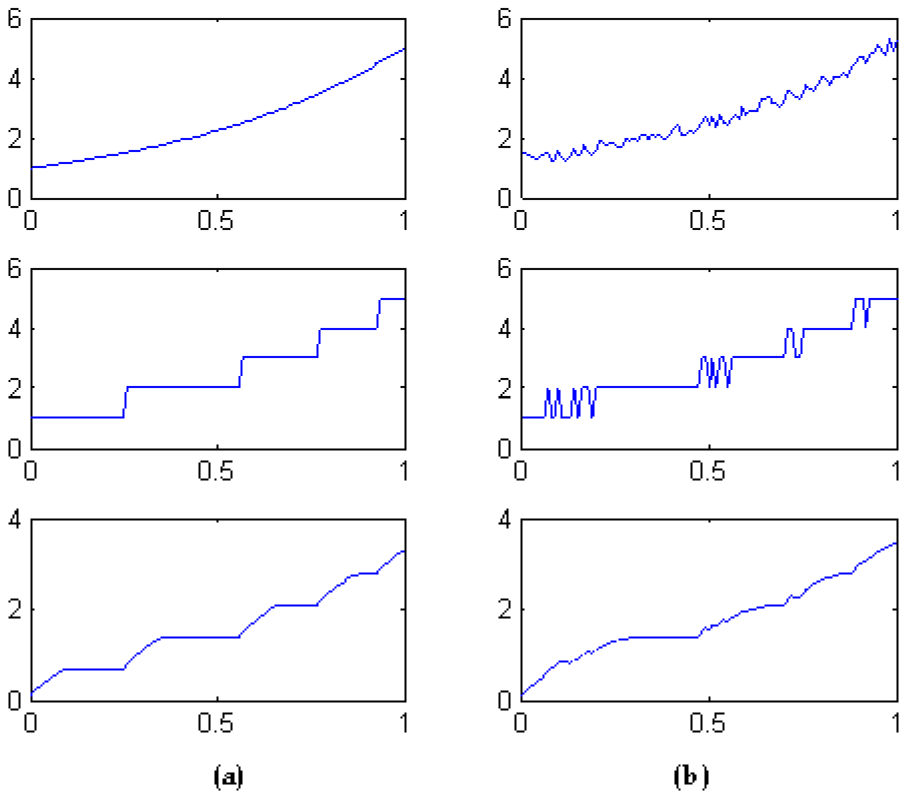
Exercise:

Problem: Dithering

Dithering is a method of decreasing the distortion of a low-frequency signal due to signal digitization [link]. Dithering works best when the sample rate is high in comparison with the rate at which the signal changes.

To see how this works, consider a slowly varying signal and its digitization, shown in [link](a). If noise is added to the original signal amplitude roughly at one half the step size, the signal will look like [link](b). If the digitized signal is passed through a resistor-capacitor circuit to smooth it out, an approximation to the original signal can be recovered. There is no theoretical limit to the accuracy possible with this method as long as the sampling rate is high enough.

Design a system to analyze the dithering technique. First, show the digitized and smoothed signal without dithering. Then, add random noise to the input signal (noise level should not exceed 50 percent of the step size of the digitized signal) and show the digitized and smoothed version. Measure the maximum and average error between the original signal and recovered signal.



Processing at One Half-Step Size: (a) From Top, the Original, Digitized and Smoothed Signal without Dithering, (b) From Top, the Noise Added, Digitized and Smoothed Signal with Dithering

---

Solution:

Insert Solution Text Here

Exercise:

Problem: Image Processing

DFT is widely used in image processing for edge detection. A digital image is a two-dimensional signal that can get stored and processed as a two-dimensional (2D) array. In the frequency domain, with the center

denoting (0,0) frequency, the center portion of this 2D array contains the low-frequency components of the 2D signal or image. The edges in the image can be extracted by removing the low-frequency components.

Read and display the image file image1.jpg provided on the book website. Then, complete the following steps:

1. Compute and display the 2D DFT of the image using the LabVIEW MathScript functions `fft2` and `fftshift`.
2. Remove the low-frequency components of the image. A user-controlled threshold can be specified to remove a varying amount of the low-frequency components.
3. Compute and display the inverse 2D DFT of the image using the LabVIEW MathScript functions `ifft2` and `fftshift`. The processed image should reflect the edges in the original image.

Solution:

Insert Solution Text Here

Exercise:

Problem: DTMF Decoder

Design a decoder VI for the DTMF system described in Telephone Signal section. The VI should be capable of reading the touchtone signal as its input and display the corresponding decoded key number as its output.

Solution:

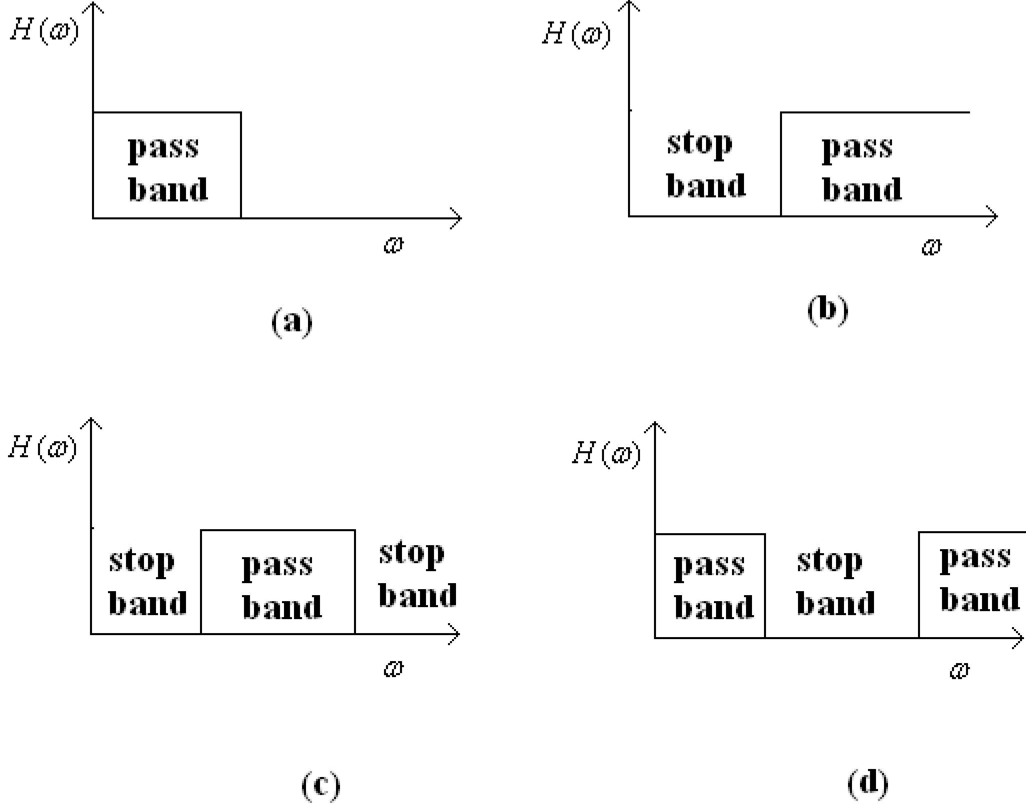
Insert Solution Text Here

Analysis of Analog and Digital Systems

In the previous labs, different mathematical transformation tools to represent analog and discrete signals were examined. This final lab builds on the knowledge gained in the previous labs to show how to use these tools to perform signal processing.

Analog Filtering

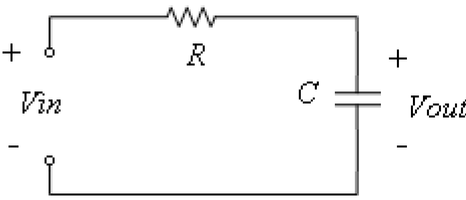
Analog filters are defined over a continuous range of frequencies. Four basic kinds of analog filters are lowpass, highpass, bandpass and bandstop. [link] shows the ideal characteristics of these filters. In the noise removal example of Lab 5 , an ideal lowpass filter was used to remove high-frequency noise. However, the ideal characteristics are not physically realizable and actual filters can only approximate the ideal characteristics. The RC series circuit analyzed in Lab 3 and Lab 4 is a simple example of an analog lowpass filter.



Characteristics of Ideal Filters (a) Lowpass, (b) Highpass, (c) Bandpass, (d) Bandstop

The voltage output for the circuit shown in [link] is given by [link]:
Equation:

$$V_{\text{out}} = \frac{1/(j\omega C)}{R + 1/(j\omega C)} V_{\text{in}}$$



RC Series Circuit Used
as Analog Lowpass Filter

The magnitude and phase response can be easily found to be [link]:

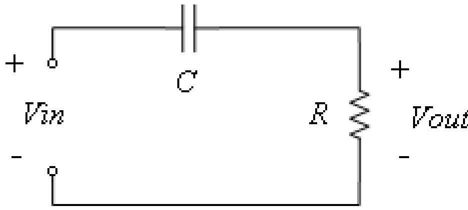
Equation:

$$\mid H(\omega) \mid = \mid \frac{V_{\text{out}}}{V_{\text{in}}} \mid = \frac{1}{\sqrt{1 + \omega^2 R^2 C^2}}$$

Equation:

$$\angle H(\omega) = \arg\left(\frac{V_{\text{out}}}{V_{\text{in}}}\right) = -\tan^{-1}(\omega \text{RC})$$

If the positions of R and C are interchanged, a simple analog highpass filter is obtained as shown in [link].



RC Series Circuit Used as
Analog Highpass Filter

The voltage output for this circuit is given by
Equation:

$$V_{\text{out}} = \frac{R}{R + 1/(j\omega C)} V_{\text{in}}$$

The corresponding magnitude and phase responses are
Equation:

$$\mid H(\omega) \mid = \mid \frac{V_{\text{out}}}{V_{\text{in}}} \mid = \frac{1}{\sqrt{1 + 1/(\omega^2 R^2 C^2)}}$$

Equation:

$$\angle H(\omega) = \arg\left(\frac{V_{\text{out}}}{V_{\text{in}}}\right) = \tan^{-1}\left(\frac{1}{\omega \text{RC}}\right)$$

Digital Filtering

Digital signal filtering is a fundamental concept in digital signal processing. Two basic kinds of digital filters that are widely used are FIR and IIR:

FIR (finite impulse response) – filters having finite unit sample responses

IIR (infinite impulse response) – filters having infinite unit sample responses

Unit sample response denotes the output in response to a unit input signal. It is common to express digital filters in the form of difference equations. In this form, an FIR filter is expressed as
Equation:

$$y[n] = \sum_{k=0}^{N} b_k x[n-k]$$

where b's denote the filter coefficients and the filter order. As described by this equation, an FIR filter uses a current input $x[n]$ and a number of previous inputs $x[n-k]$ to generate a current output $y[n]$.

The difference equation of an IIR filter is given by
Equation:

$$y[n] = \sum_{k=0}^{N} b_k x[n-k] - \sum_{k=1}^{M} a_k y[n-k]$$

where b's and a's denote the filter coefficients and N and M the number of zeros and poles, respectively. As indicated by Equation (8), an IIR filter uses a number of previous outputs $y[n-k]$ as well as a current and a number of previous inputs to generate a current output $y[n]$.

In general, as compared to IIR filters, FIR filters require less precision and are computationally more stable. Table 1 lists some of the differences between FIR and IIR filters. For the theoretical details on these differences, refer to [link].

Attribute	**FIR filter**	**IIR filter**
Stability	Always stable	Conditionally stable
Computational complexity	More operations	Fewer operations
Precision	Less coefficient precision required	Higher coefficient precision required

where b’s and a’s denote the filter coefficients and and the number of zeros and poles, respectively. As indicated by Equation (8), an IIR filter uses a number of previous outputs $y[n-k]$ as well as a current and a number of previous inputs to generate a current output $y[n]$.

Equation:

$$H(\omega)=\frac{b_0+b_1e^{-j\omega}+...+b_Ne^{-\mathrm{jN}\omega}}{1+a_1e^{-j\omega}+...+a_Me^{-\mathrm{jM}\omega}}$$

It is well-known that a direct-form implementation expressed by Equation (9) is sensitive, in terms of stability, to coefficient quantization errors. Noting that the second-order cascade form produces a more response that is more resistant to quantization noise [link], the above transfer function is often written and implemented as follows:

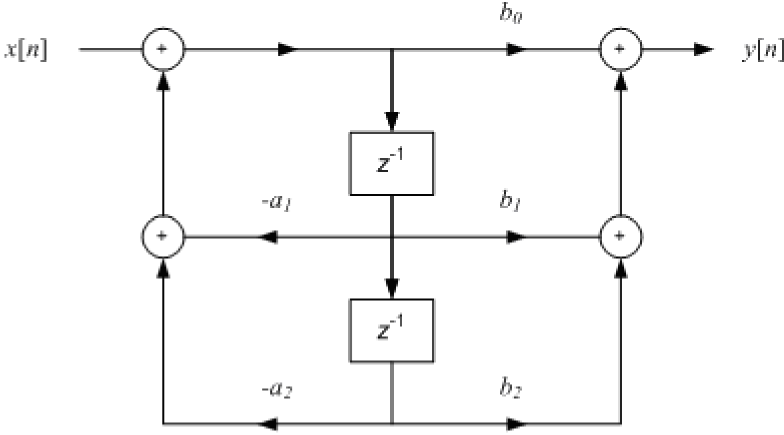
Equation:

$$H(\omega)=\prod_{k=1}^{N_s}\frac{b_{0\mathrm{k}}+b_{1\mathrm{k}}e^{-j\omega}+b_{2\mathrm{k}}e^{-j2\omega}}{1+a_{1\mathrm{k}}e^{-j\omega}+a_{2\mathrm{k}}e^{-j2\omega}}$$

where $N_s=\lfloor N/2\rfloor$ and $\lfloor.\rfloor$ represents the largest integer less than or equal to the argument. This serial or cascaded structure is illustrated in [link].

Cascaded Filter Stages

Each second-order filter is considered to be of direct-form II, shown in [link], for its memory efficiency. One can implement each second-order filter in software. Normally, digital filters are implemented in software, but one can also implement them in hardware by using digital circuit adders and shifters.



Second Order Direct-Form II

Lab 7: System Response, Analog and Digital Filters

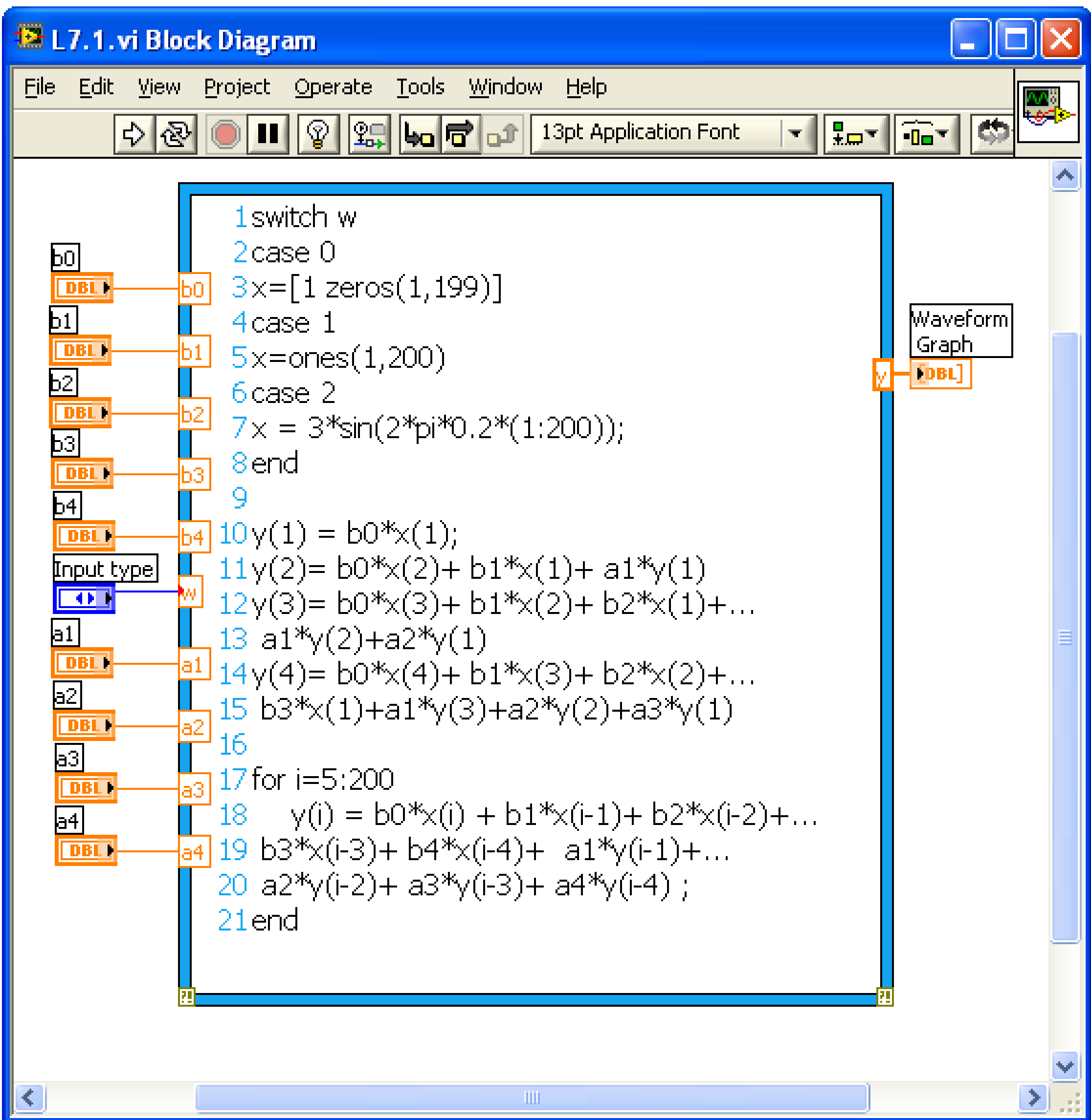
Response of Discrete-Time Systems

This lab involves analyzing the response of discrete-time systems. Responses are calculated for three different kinds of inputs; impulse, step and sine. [link] shows the completed block diagram. Connect the input variable w to an **Enum Control** so that an input type (impulse, step or sine) can be selected. The response of this system to any discrete-time input $x[n]$ can be written as

Equation:

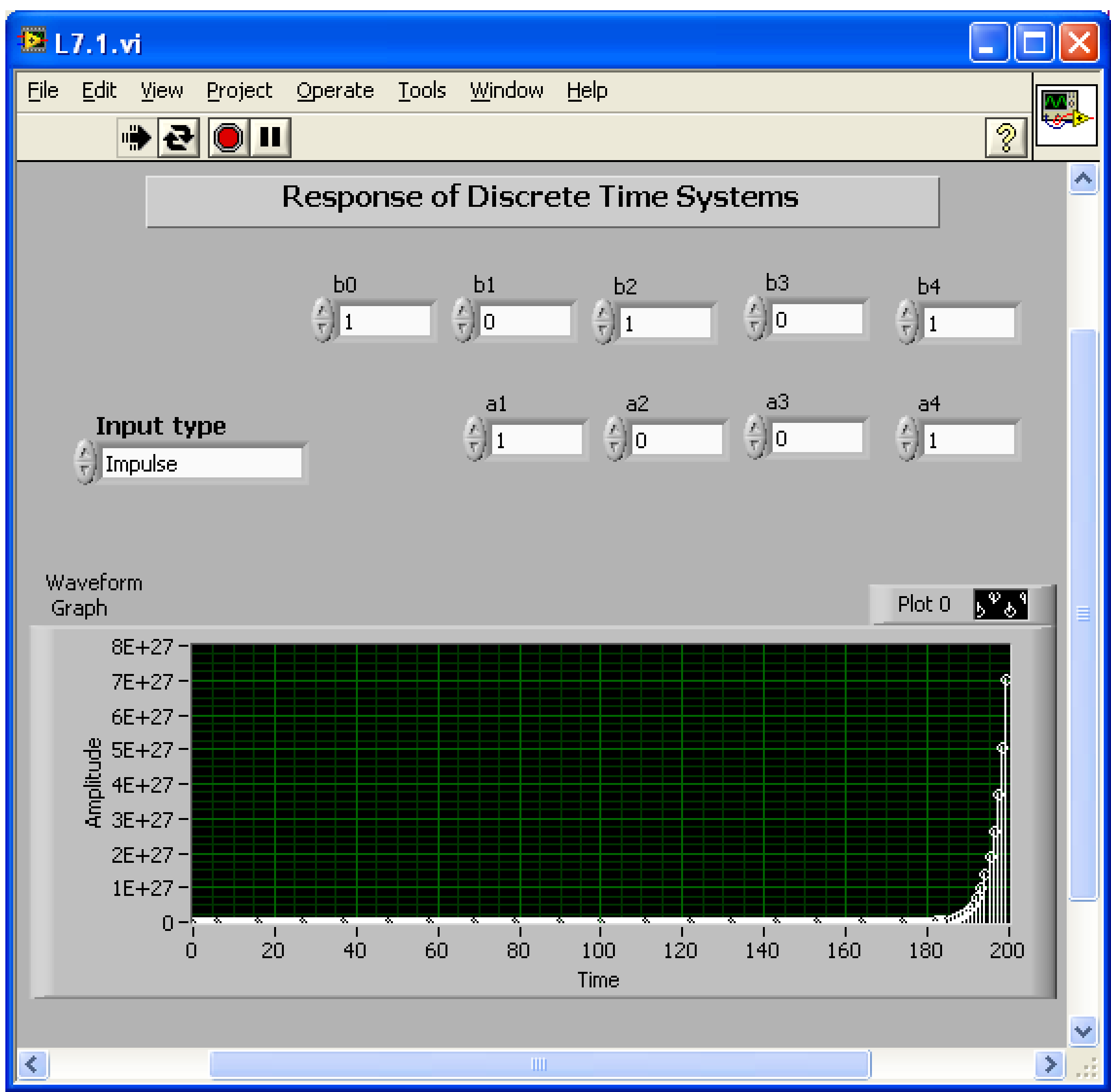
$$y[n] = \sum_{i} b_i x[n-i] + \sum_{i} a_i y[n-i]$$

For this example, consider five b’s and four a’s. The system output is displayed using a waveform graph.



Block Diagram of a Discrete-Time System

[link] shows the front panel of the above system. The front panel can be used to interactively select the input type and set the coefficients a and b. The system response for a particular type of input (impulse, step or sine) is shown in the waveform graph.



Front Panel of a Discrete-Time System

Square Root

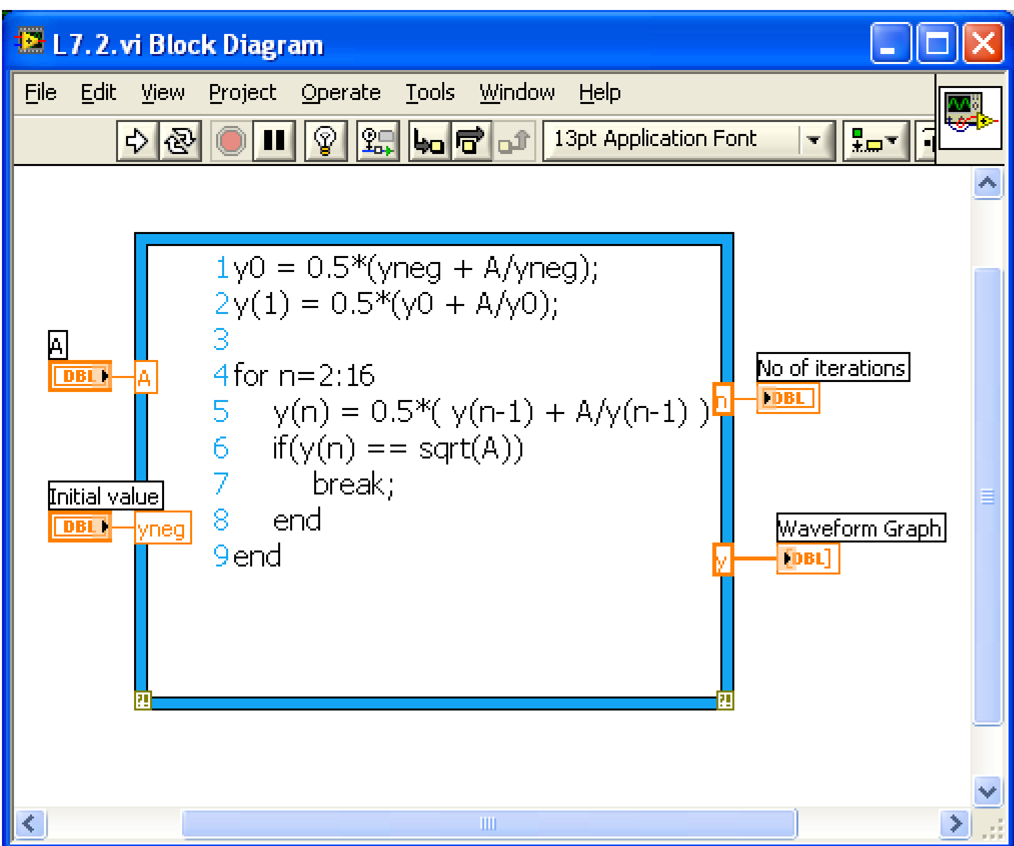
As another example of discrete-time systems, let us consider taking the square root of an integer number. Often computers and calculators compute the square root of a positive number A using the following recursive equation:

Equation:

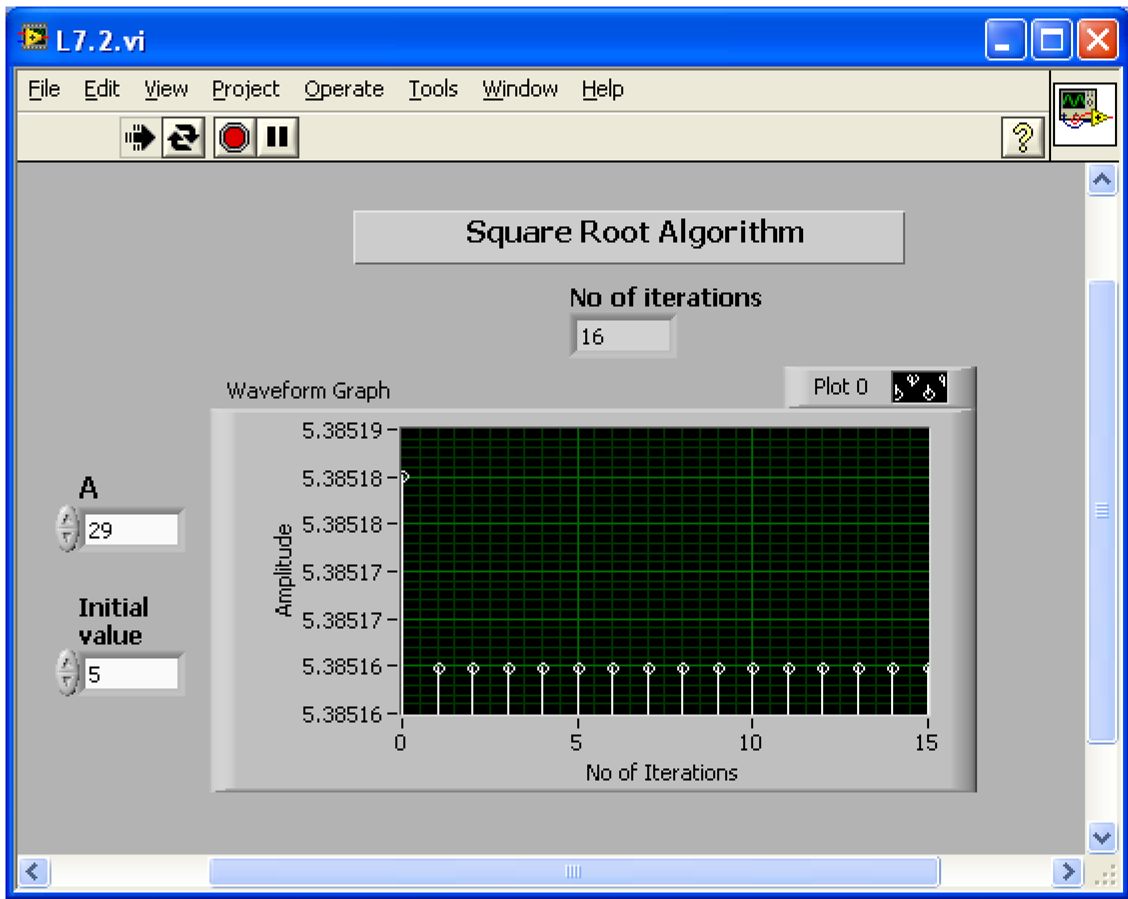
$$y[n] = \frac{1}{2}\left(y[n-1] + \frac{x[n]}{y[n-1]}\right)$$

If the input $x[n]$ to this equation is set as a step function of amplitude A, then $y[n]$ converges to the square root of A after several iterations.

[link] shows the block diagram for a square root computation system. The number of iterations required to converge to the true value is shown in the output. The initial condition Initial value is set as a control. [link] shows the corresponding front panel.



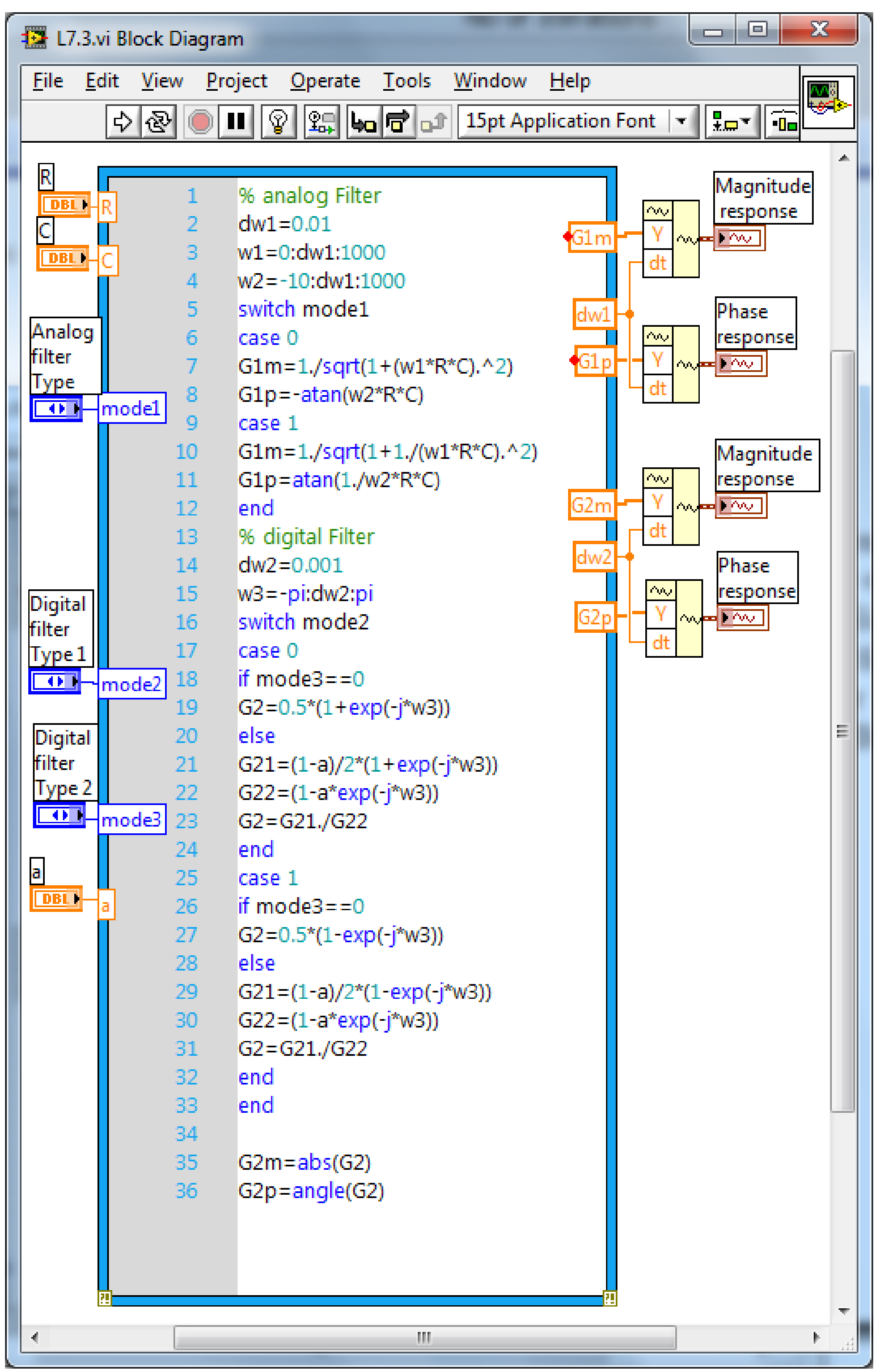
Block Diagram of a Square Root Computation System



Front Panel of a Square Root Computation System

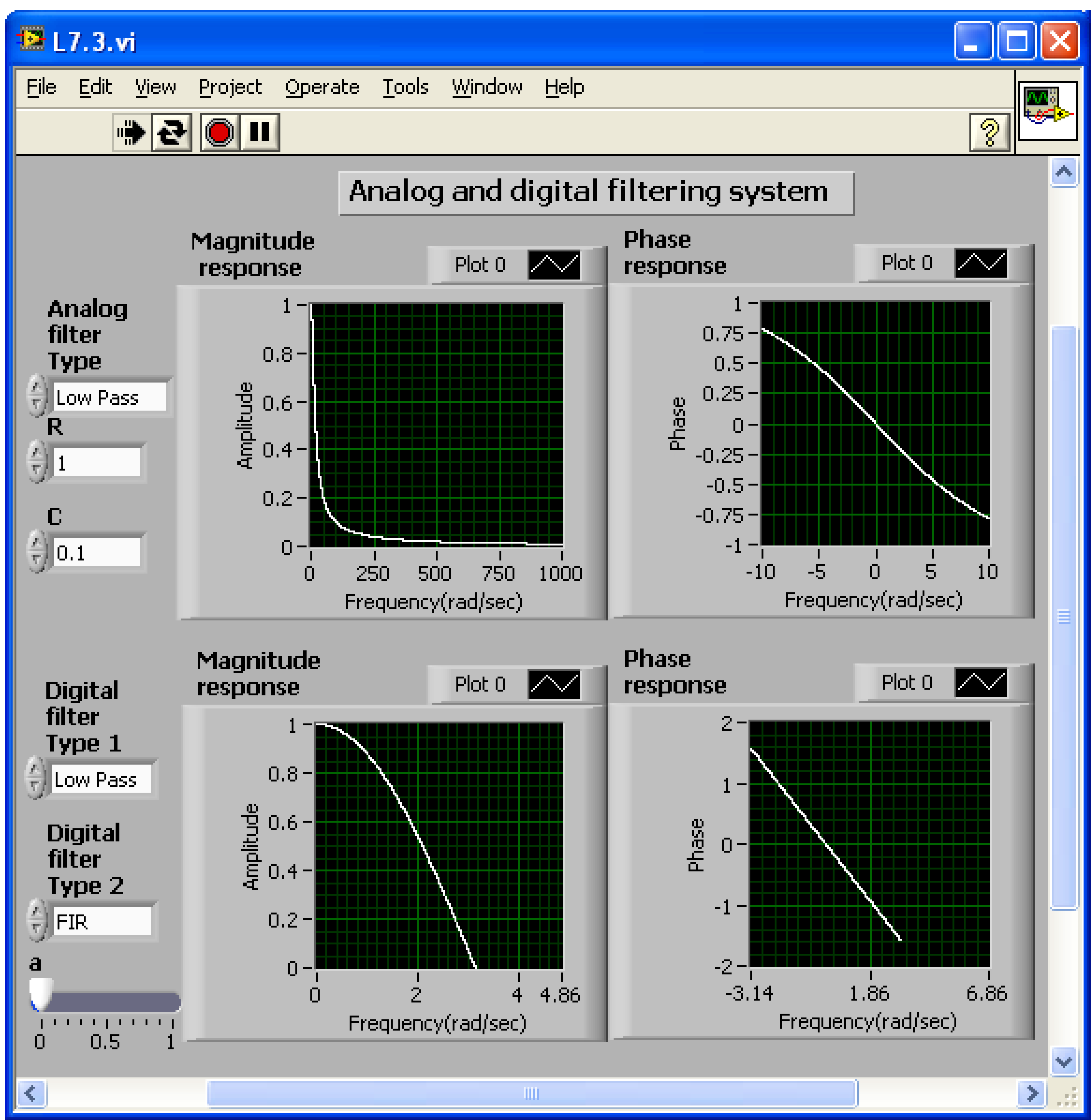
Analog and Digital Filtering

In this section, let us examine a basic analog and digital filtering example by implementing a lowpass and a highpass filter in the analog and digital domains, respectively. [link] shows the completed block diagram of the filtering system. For analog approximation of the signals, use a higher sampling rate (dw1=0.01). To detect whether the filtering is lowpass or highpass, use the Enum Control Analog filter type. Calculate the magnitude and phase response of these filters using equations provided in Chapter 7 for analog and digital filters. Set the values of R and C as controls, and display the responses using a Build Waveform function and a waveform graph.

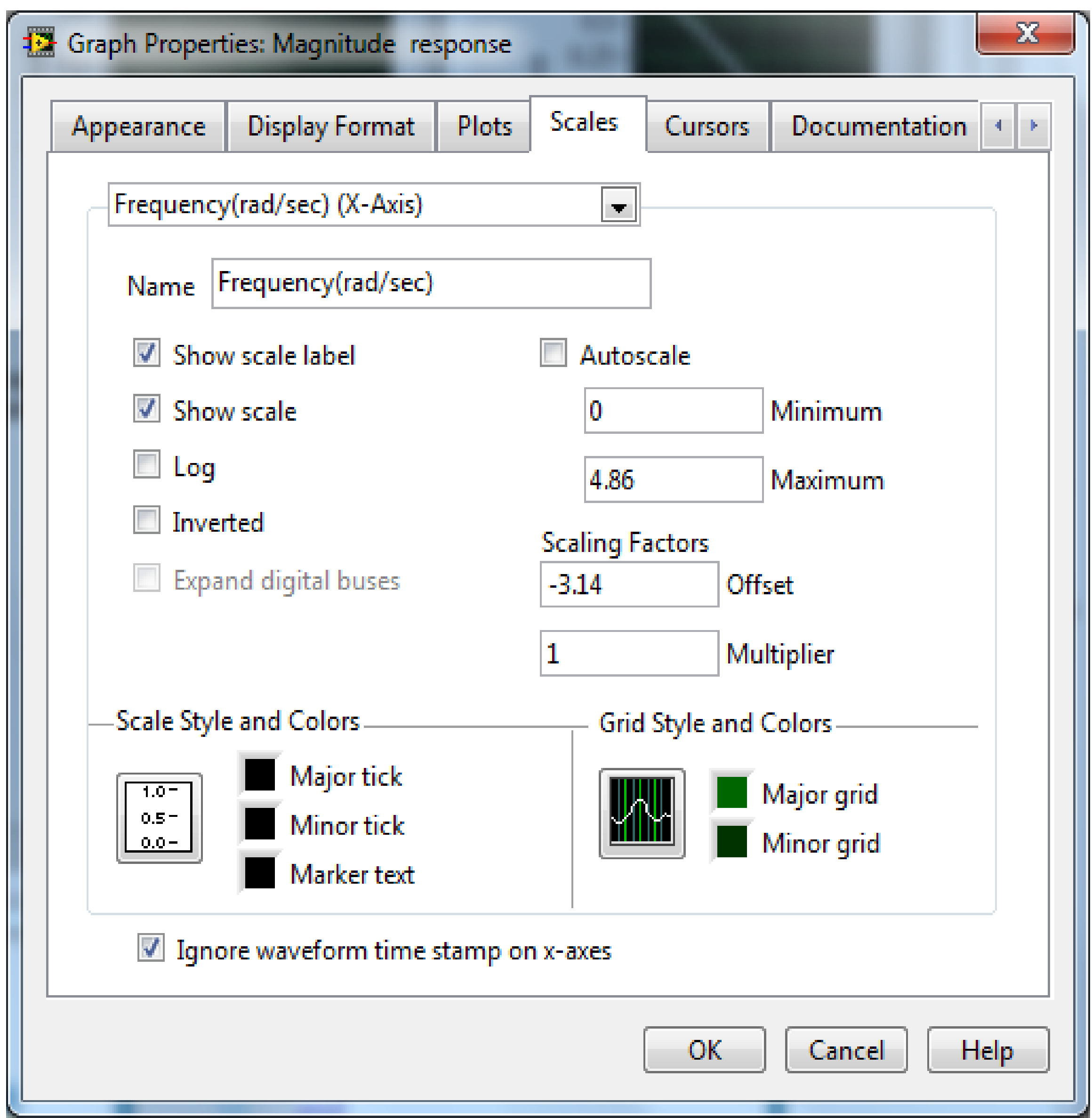


Block Diagram of an Analog and Digital Filtering System

For the digital case, use a lower sampling rate (dw2=0.001). With the Enum controls Digital filter type 1 and Digital filter type 2, select lowpass or highpass and FIR or IIR filter type. Use a Build Waveform function and a waveform graph to display the magnitude and phase responses of the digital filters. [link] shows the front panel of this filtering system. For a better view of magnitude response of the digital filter, set the properties of the waveform graph as shown in [link] .



Front Panel of an Analog and Digital Filtering System



Graph properties of magnitude response of digital filter

Lab Exercises

Exercise:

Problem: Bandpass and Bandstop Filters

Use the lowpass and highpass filters (both analog and digital) described in Analog and Digital Filtering section to construct bandpass and bandstop filters. The bandpass filter should be able to pass signals from 50 to 200 Hz and the bandstop filter should be able to stop signals from 150 to 400 Hz. Determine the values of R and C required for this analog filter design. Also, determine the values of the coefficients required for an equivalent IIR digital filter design.

Solution:

Insert Solution Text Here

Exercise:

Problem: Noise Reduction

Use an analog lowpass filter to remove the high-frequency noise described in Noise Reduction example of Lab 5. Repeat using a digital lowpass filter.

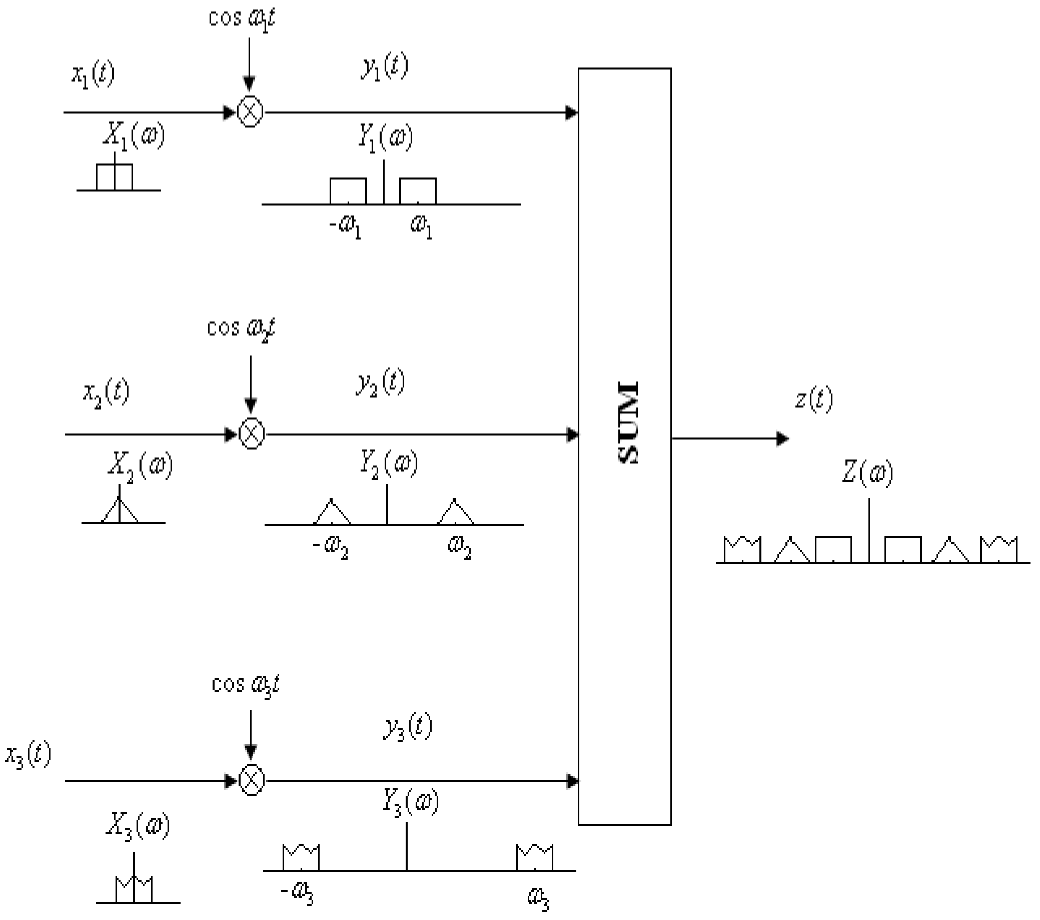
Solution:

Insert Solution Text Here

Exercise:

Problem: Frequency Division Multiplexing (FDM)

FDM is widely used in digital communication to simultaneously transmit multiple signals over a single wideband channel (for details, refer to [link]). For FDM communication, individual signals are multiplied with different carriers to avoid overlaps in the frequency domain. Their time domain processing and corresponding frequency spectrums are shown in [link]. Build a VI to implement an FDM communication system for three signals $x_1(t)$,$x_2(t)$and $x_3(t)$. Use the files echo_1.wav and firetrucksiren.wav on the book website and a random noise with a frequency range of 20 Hz to 20 kHz to serve as these signals.



FDM Communication System

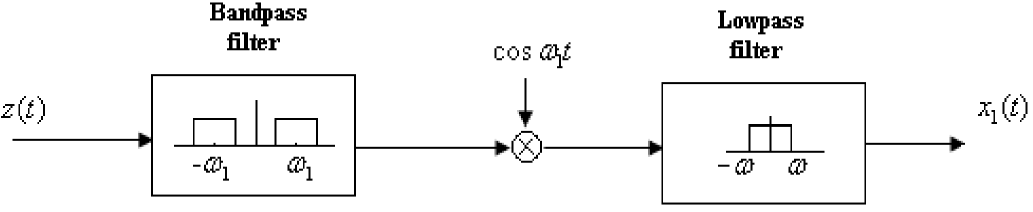
Solution:

Insert Solution Text Here

Exercise:

Problem: FDM Detector

Build a VI to implement an FDM detector system for detecting the signal $x_1(t)$ as shown in [link].



FDM Detector

Solution:

Insert Solution Text Here

References